(19)日本国特許庁（ＪＰ）　(12)公開特許公報（Ａ）　(11)特許出願公開番号

特開平11－237989

(43)公開日　平成11年(1999)８月31日

(51)Int.Cl.[6]		識別記号	FI		
Ｇ０６Ｆ	9/45		Ｇ０６Ｆ	9/44	３２２Ａ
	9/445			9/06	４２０Ｂ

審査請求　未請求　請求項の数22　ＯＬ　外国語出願　（全 56 頁）

(21)出願番号　特願平10－319738

(22)出願日　平成10年(1998)10月６日

(31)優先権主張番号　０８／９４４７３５

(32)優先日　1997年10月６日

(33)優先権主張国　米国（ＵＳ）

(71)出願人　595034134
サン・マイクロシステムズ・インコーポレイテッド
Ｓｕｎ　Ｍｉｃｒｏｓｙｓｔｅｍｓ，　Ｉｎｃ．
アメリカ合衆国　カリフォルニア州　94303　パロ　アルト　サン　アントニオ　ロード　901

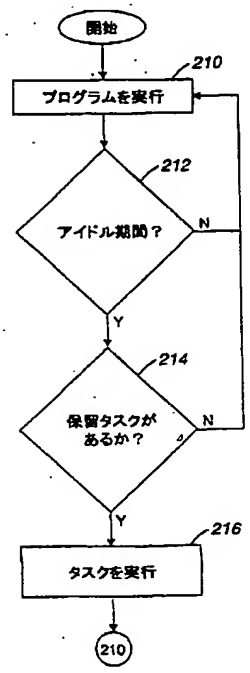
(72)発明者　ウールズ　ホルツル
アメリカ合衆国，　カリフォルニア州，　ゴリータ，　ダヴェンポート　ロード　7220，　ナンバー105

(74)代理人　弁理士　長谷川　芳樹　（外３名）

最終頁に続く

(54)【発明の名称】　休止中にバイトコード最適化を実行する方法及び装置

(57)【要約】
【課題】　コンピュータプログラムにおけるメソッドの動的コンパイルのみかけ上の効率を改善する。
【解決手段】　コンピュータプログラムの実行におけるアイドル期間中にメソッドに付随するバイトコードを動的コンパイルする方法および装置が開示される。ここで述べる方法は、解釈済バイトコードおよびコンパイル済バイトコードの双方を実行するように構成されたコンピュータシステム内で使用するのに特に適している。一部の態様では、動的コンパイルすべきメソッドが一つ以上のリスト中で参照される。このリストは、最高優先順位メソッドのコンパイルを最初に促進するように優先順位付けすることができる。ある態様では、一対のコンパイルリストが用意される。これらのコンパイルリストのうちの最初の一つはコンピュータプログラムの処理に先立って生成され、他方はコンピュータプログラムの処理中に生成される。



1

【特許請求の範囲】

【請求項１】　コンピュータプログラムの処理におけるアイドル期間中にメソッドを動的コンパイルするためにコンピュータによって実施される方法であって、

前記アイドル期間を識別するステップと、

複数のメソッドから選択される第１のメソッドを前記アイドル期間中に識別するステップであって、前記複数のメソッドは、前記コンピュータプログラムに含まれており、このコンピュータプログラムは、解釈済プログラムコードおよびコンパイル済プログラムコードの双方を実行するように構成されているステップと、

前記第１メソッドのコンパイルを初期化するステップであって、この第１メソッドのコンパイルの初期化が前記アイドル期間中に行われるようになっているステップと、を備える方法。

【請求項２】　前記複数のメソッドは、前記コンピュータプログラムに付随する第１のリスト中で参照され、前記第１メソッドは、前記第１リストに付随する最高優先順位メソッドである、請求項１記載のコンピュータによって実施される方法。

【請求項３】　前記複数のメソッドから選択される第１の一組のメソッドは第１のリストに付随し、前記複数のメソッドから選択される第２の一組のメソッドは第２のリストに付随し、前記第１リストは前記コンピュータプログラムの処理中に生成され、前記第２リストは前記コンピュータプログラムの処理前に生成され、

前記第１メソッドを識別するステップは、前記第１リストから最高優先順位メソッドとして前記第１メソッドを選択するステップを含んでいる、請求項１記載のコンピュータによって実施される方法。

【請求項４】　前記複数のメソッドから選択される第１の一組のメソッドは第１のリストに付随し、前記複数のメソッドから選択される第２の一組のメソッドは第２のリストに付随し、前記第１リストは前記コンピュータプログラムの処理中に生成され、前記第２リストは前記コンピュータプログラムの処理前に生成され、

前記第１メソッドを識別するステップは、前記第２リストから最高優先順位メソッドとして前記第１メソッドを選択するステップを含んでいる、請求項１記載のコンピュータによって実施される方法。

【請求項５】　前記第１メソッドのコンパイルの初期化後に前記コンピュータプログラムによって割込みが受信されたどうかを判断するステップと、

割込みが受信されたと判断された場合に所定時間にわたって前記第１メソッドコンパイルを継続するステップと、

前記第１メソッドのコンパイルが前記所定時間の経過後に完了したかどうかを判断するステップと、

前記第１メソッドのコンパイルが前記所定時間の経過後に完了していないと判断された場合に前記第１メソッドのコンパイルを打ち切るステップと、を更に備える請求項１～４のいずれかに記載のコンピュータによって実施される方法。

2

【請求項６】　コンピュータプログラムの全体処理における低アクティビティ期間中にメソッドを動的コンパイルするためにコンピュータによって実施される方法であって、

前記低アクティビティ期間を識別するステップと、

複数のメソッドから選択される第１のメソッドを前記低アクティビティ期間中に識別するステップであって、前記複数のメソッドは前記コンピュータプログラムに含まれているステップと、

前記第１メソッドがコンパイルされたかどうかを判断するステップであって、この判断が前記低アクティビティ期間中になされるステップと、

前記第１メソッドがコンパイルされていないと判断された場合に、前記低アクティビティ期間中に前記第１メソッドのコンパイルを初期化するステップと、を備える方法。

【請求項７】　前記第１メソッドがコンパイルされたと判断された場合に、前記複数のメソッドから選択される第２のメソッドを前記低アクティビティ期間中に識別するステップと、

前記第２メソッドがコンパイルされたかどうかを判断するステップであって、前記第２メソッドがコンパイルされたかどうかの判断が前記低アクティビティ期間中に行われるステップと、

前記第２メソッドがコンパイルされていないと判断された場合に、前記低アクティビティ期間中に前記第２メソッドのコンパイルを初期化するステップと、を更に備える請求項６記載のコンピュータによって実施される方法。

【請求項８】　前記複数のメソッドの各々に対して優先順位値を決定するステップを更に備え、前記第１メソッドが前記第１メソッドに対する前記優先順位値に基づいて識別される請求項６または７記載のコンピュータによって実施される方法。

【請求項９】　前記複数のメソッドの各々に対して前記優先順位値を決定するステップは、前記複数のメソッドの各々に関連付けられた呼出しカウンタを前記低アクティビティ期間中に処理するステップを含んでいる、請求項８記載のコンピュータによって実施される方法。

【請求項１０】　前記複数のメソッドの各々に対して前記優先順位値を決定するステップは、前記複数のメソッドの各々に対するコンパイル時間を前記低アクティビティ期間中に推定するステップを含んでいる、請求項８記載のコンピュータによって実施される方法。

【請求項１１】　前記複数のメソッドから選択される第１の一組のメソッドは第１のリストに付随し、前記複数のメソッドから選択される第２の一組のメソッドは第２

のリストに付随し、前記第1リストは前記コンピュータプログラムの処理中に生成され、前記第2リストは前記コンピュータプログラムの処理前に生成され、

前記第1メソッドを識別するステップは、前記第1リストから最高優先順位メソッドとして前記第1メソッドを選択するステップを含んでいる、請求項6記載のコンピュータによって実施される方法。

【請求項12】 前記第1メソッドのコンパイルの初期化後に前記コンピュータプログラムによって割込みが受信されたかどうかを判断するステップと、

割込みが受信されたと判断された場合に所定時間にわたって前記第1メソッドのコンパイルを継続するステップと、

前記第1メソッドのコンパイルが前記所定時間の経過後に完了したかどうかを判断するステップと、

前記第1メソッドのコンパイルが前記所定時間の経過後に完了していないと判断された場合に前記第1メソッドのコンパイルを打ち切るステップと、を更に備える請求項6～11のいずれかに記載のコンピュータによって実施される方法。

【請求項13】 割込みが受信されていないと判断された場合に前記第1メソッドのコンパイルを完了するステップと、

複数のメソッドから選択される第2のメソッドを前記低アクティビティ期間中に識別するステップと、

前記第2メソッドが前記低アクティビティ期間中にコンパイルされたかどうかを判断するステップと、

前記第2メソッドがコンパイルされていないと判断された場合に、前記低アクティビティ期間中に前記第2メソッドのコンパイルを初期化するステップと、を更に備える請求項12記載のコンピュータによって実施される方法。

【請求項14】 コンピュータプログラムの処理におけるアイドル期間中に遅延タスクを動的に処理するためにコンピュータによって実施される方法であって、

前記コンピュータプログラムの処理におけるアイドル期間を識別するステップと、

複数のタスクから選択される第1のタスクを前記コンピュータプログラムの処理におけるアイドル期間中に識別するステップであって、前記複数のタスクは、前記コンピュータプログラムに含まれており、このコンピュータプログラムは、解釈済プログラムコードおよびコンパイル済プログラムコードの双方を実行するように構成されているステップと、

前記第1タスクを開始するステップであって、前記第1タスクの開始が前記アイドル期間中に行われるようになっているステップと、を備える方法。

【請求項15】 割込みが受信された場合に前記第1タスクを完了するステップを更に備える請求項14記載のコンピュータによって実施される方法。

【請求項16】 割込みが受信された場合に前記第1タスクを中断するステップを更に備える請求項14記載のコンピュータによって実施される方法。

【請求項17】 コンピュータプログラムの処理における低アクティビティ期間中にバイトコード化メソッドを動的コンパイルするコンピュータシステムであって、前記メソッドは前記コンピュータプログラムに付随しており、

前記低アクティビティ期間を識別する機構と、

複数のメソッドから選択される第1のメソッドを前記低アクティビティ期間中に識別する機構と、

前記低アクティビティ期間中に前記第1メソッドのコンパイルを初期化するコンパイラと、を備えるコンピュータシステム。

【請求項18】 前記複数のメソッドの第1の一組を参照するように構成され、前記コンピュータプログラムに付随する第1のリストであって、前記第1メソッドがこの第1リストに付随する最高優先順位メソッドとなっている第1リストと、

前記複数のメソッドの第2の一組を参照するように構成された第2のリストであって、前記コンピュータプログラムの処理前に生成される第2リストと、を更に備え、

前記第1リストが前記コンピュータプログラムの処理中に生成される請求項17記載のコンピュータシステム。

【請求項19】 前記第1メソッドのコンパイル中に割込み信号を受信するように構成され、前記コンパイラと通信する割込みプロセッサであって、前記コンパイラは、この割込みプロセッサからの信号に応答して前記第1メソッドのコンパイルを所定時間にわたって継続するように更に構成されている、割込みプロセッサと、

前記コンパイラと通信する打切り機構であって、前記第1メソッドのコンパイルを打ち切るように構成された打切り機構と、を更に備える請求項17記載のコンピュータシステム。

【請求項20】 前記第1メソッドがコンパイルされる時点を識別するように構成された診断機構を更に備える請求項17～19のいずれか記載のコンピュータシステム。

【請求項21】 コンピュータプログラムの処理におけるアイドル期間中にメソッドを動的コンパイルするように構成されたコンピュータプログラムコード装置（computer program code device）を含むコンピュータ読取り可能な媒体であって、

前記アイドル期間を識別するコンピュータプログラムコード装置と、

複数のメソッドから選択される第1のメソッドを前記アイドル期間中に識別するコンピュータプログラムコード装置であって、前記複数のメソッドが前記コンピュータプログラムに含まれているコンピュータプログラムコード装置と、

前記アイドル期間中に前記第１メソッドのコンパイルを初期化するコンピュータプログラムコード装置と、を備えるコンピュータ読取り可能な媒体。

【請求項２２】　前記コンピュータプログラムコード装置が搬送波に具体化されている請求項２１記載のコンピュータ読取り可能な媒体。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】本発明は、全般にソフトウェアアプリケーションの実行を最適化する方法および装置に関する。本発明は、特に、コンピュータプログラムの全体的実行における休止中にコンパイルや他のアクティビティを実行してコンピュータシステム資源の使用を最適化する方法および装置に関する。

【０００２】

【従来の技術】コンピュータシステムは、しばしばコンピュータシステムのネットワーク（例えば、ローカルエリアネットワークやイントラネットやインターネット）を介して連結されており、これらのコンピュータシステムがソフトウェアアプリケーションなどの資源を共有できるようになっている。一般に、ソフトウェアアプリケーション、あるいはコンピュータプログラムは、異なるコンピュータシステムに異なるフォーマットで配信されることがある。これは、各コンピュータシステムが、ソフトウェアアプリケーションが特定のフォーマットであることを要求するという事実によるものである。この他に、一つの形式のコンピュータプログラムを多くの異なるコンピュータシステムが利用できるようにするために、コンピュータプログラムが、機械独立形式、すなわちバイトコードとしてコンピュータシステムに配信される場合もある。

【０００３】コンピュータプログラムを機械独立形式で配信すると、プログラムを直接解釈したり、あるいはプログラムを機械依存コード、すなわち「マシンコード」に翻訳することができる。直接解釈されるプログラムは、マシンコードに翻訳されるプログラムよりもコンピュータシステム内で占めるスペースが少ない。しかしながら、直接解釈されるプログラムは、たいていの場合、マシンコードに翻訳されるプログラムよりも実行速度が遅い。このため、コンピュータプログラムをマシンコードに翻訳する代わりにコンピュータプログラムを直接解釈するかどうかは、多くの場合、実行速度に対するスペースの相対的な重要度に基づいて判断される。

【０００４】

【発明が解決しようとする課題】コンパイルへの一つのアプローチは、アクティブに動作中のプログラム内のメソッドが実行のために最初に呼び出されるときにそのメソッドをコンパイルすることである。このアプローチは、しばしば動的コンパイルまたは「実行時（runtime）」コンパイルと呼ばれる。動的コンパイルに伴う一つの問題点は、プログラムに伴うコンパイルオーバヘッドが過大になる場合があることである。すなわち、比較的大きな数のメソッドをほぼ同時にコンパイルしなければならない場合、プログラムのコンパイルに伴うオーバヘッドが、プログラムの全体的な実行に悪影響を与えるレベルとなる場合がある。一般に、コンパイルにシステム資源を使用しすぎると、プログラムの実行が低効率で遅くなることがある。更に、アクティブプログラム実行中にメソッドをコンパイルしているコンパイラによって消費された時間は、ユーザから見えるような休止をプログラムの実行に導く可能性がある。例えば、プログラムのコンパイルオーバヘッドが高い間にユーザがプログラムにコマンドを入力すると、ユーザ入力に対する応答に遅延が生じる場合がある。このような応答の遅延は、ユーザを苛立たせることがある。従って、コンピュータプログラムにおけるメソッドの動的コンパイルの外見上の効率を改善する機構が望まれる。

【０００５】

【課題を解決するための手段】本発明の上記の目的やその他の目的を達成するため、コンピュータプログラムの実行におけるアイドル期間中にメソッドを動的コンパイルする方法および装置を開示する。ここに記載する方法は、解釈済バイトコードおよびコンパイル済バイトコードの双方を実行するように構成されたコンピュータシステムで使用するのに特に適している。本発明の一態様によれば、コンピュータプログラムの処理におけるアイドル期間または低オーバヘッド期間が識別される。この後、コンパイル用に識別された一つ以上のメソッドがこの識別されたアイドル期間中に動的コンパイルされる。一部の態様では、動的コンパイルされるべきメソッドが一つ以上のリスト中で参照される。これらのリストは、最も高い優先順位のメソッドのコンパイルを最初に促進するように優先順位付けされていてもよい。ある態様では、一対のコンパイルリストが設けられる。これらのコンパイルリストの最初の一つは、コンピュータプログラムを処理する前に作成されるデータベースリストであり、他方は、コンピュータプログラムの処理中に作成される実行リストである。

【０００６】ある態様では、メソッドのコンパイル中に割込みが受信されると、所定の時間にわたってコンパイルの継続が可能になる。コンパイルがこの所定時間中に完了しない場合、コンパイルは打ち切られる。

【０００７】本発明の更に別の態様では、コンピュータプログラムの処理における低アクティビティ期間中にバイトコード化メソッドを動的コンパイルするコンピュータシステムは、低アクティビティ期間を識別する機構を含んでいる。このシステムは、低アクティビティ期間中のコンパイルのためのメソッドを識別する機構を更に含んでいる。コンパイラは、低アクティビティ期間中に識別済メソッドのコンパイルを初期化するように構成され

ている。ある態様では、このシステムは、識別済メソッドのコンパイル中に受信された割込み信号を処理するように構成された割込みプロセッサ、および必要なときにメソッドのコンパイルを打ち切るアボート機構も含んでいる。

【０００８】本発明の上記および他の利点は、以下の詳細な説明を読み、種々の添付図面を検討することにより明らかになる。

【０００９】本発明は、添付の図面とともに以下の説明を参照することによって最も良く理解することができる。

【００１０】

【発明の実施の形態】上述のように、プログラムがアクティブに動作している間にバイトコード化プログラムに伴うメソッドを動的コンパイルすることにより、プログラムの動作が非効率的になる場合がある。この非効率性は、コンパイラによって利用可能なコンピュータシステム資源の過度の使用、すなわちコンパイルオーバヘッドという結果を生み出すことがある。アクティブプログラム実行中にメソッドを動的コンパイルしているコンパイラによって費やされる時間は、プログラムの実行に遅延を導く場合がある。このような遅延をユーザが知覚できる場合、特に遅延の長さが比較的長い（例えば、遅延の長さが約２００ミリ秒を超える）場合、このような遅延は容認できないものとみなされることがある。

【００１１】このようなプログラムの性能の改善に対する一つのアプローチは、解釈済バイトコードおよび動的コンパイル済バイトコードの双方の混在を可能にするとである。このようなシステムでは、特定のメソッドのコンパイルをいつ行うのが効率的かに関する判断を依然として行わなければならない。本発明では、コンパイルオーバヘッドが比較的高いときにコンピュータプログラム中のメソッドの動的コンパイルを遅延させる方法および装置が記載される。特に、コンパイルオーバヘッドが高すぎると判断される場合、プログラムの全体処理に比較的低いアクティビティの期間が存在するようになるまで、メソッドの少なくとも一部の動的コンパイルを遅延させることができる。

【００１２】コンパイルが予定されている少なくとも一部のバイトコード化メソッドの動的コンパイルを遅延させることにより、コンピュータ資源をより効率良く利用することができる。コンピュータプログラムが処理中において本質的にインアクティブである間、例えばコンピュータプログラムがユーザからの入力を待っている間アイドルであるとき、プログラムの実行に伴う全体資源利用オーバヘッドは、一般に比較的低い。このような低アクティビティ期間中は、コンパイルすべきメソッドであると事前に識別されたメソッドのコンパイルを、利用可能なシステム資源を用いて実行することができる。コンピュータプログラムの計算オーバヘッドが低い間にコンパイルを実行することは、システム資源全体の効率的な使用を見越しており、通常、ユーザにとって目立つ遅延を生じることはない。

【００１３】最初に図１を参照し、バイトコードの動的コンパイルを可能にするコンピュータシステムを本発明の一実施形態に従って説明する。バイトコード１４４は、実行時にコンピュータシステム１４６への入力として与えられる。バイトコード１４４は、一般にコンピュータプログラムとして編成することができるが、通常は、メソッドなどのバーズ可能セグメント、またはルーチンに配列されている。ある実施形態では、バイトコード１４４は、仮想マシン内の翻訳時環境（compile-time environment）によって同じ仮想マシン内の実行時環境（run-time environment）、例えばコンピュータシステム１４６に提供することができる。本発明を実施可能な一つの適切な仮想マシンについは、図６を参照しながら詳細に後述する。

【００１４】バイトコード１４４がコンピュータシステム１４６に与えられた場合、バイトコード１４４をインタープリタ１４８によって処理することができる。あるいは、バイトコード１４４をコンパイラ１５０によってコンパイルして、コンパイル済バイトコード１５２を作成してもよい。一般に、バイトコード１４４は、処理のためにインタープリタ１４８およびコンパイラ１５０の双方にほぼ同時に入力されることがある。

【００１５】メソッドの各呼出しでは、メソッドがコンパイルされない場合、そのメソッドに伴うバイトコード１４４がインタープリタ１４８を用いて解釈される。ある実施形態では、一つのメソッドが解釈される回数の計測器が維持される。このような計測器は、各解釈済メソッドに含まれるカウンタであってメソッドが解釈される度に増分されるカウンタ（例えば呼出しカウンタ）を含んでいてもよい。

【００１６】メソッドが解釈される回数がしきい値、すなわち制限値を超えると、そのメソッドは、コンパイラ１５０を用いてコンパイルすることができる。解釈済バイトコード１５４は、一般にコンパイル済コードよりも実行速度が遅い、すなわち非効率的であるので、頻繁に実行されるコード（frequently executed code）１５８に含まれるメソッドを繰り返し解釈することは非効率的な場合がある。メソッドをコンパイルすることにより得られる時間節約は、コンパイルプロセスに伴うコンパイルオーバヘッドを補償する可能性が高いので、一般に、頻繁実行コード１５８をコンパイルすることで、頻繁実行コード１５８に具現化されたメソッドをより効率良く実行することができる。

【００１７】実行時の間、コンパイラ１５０を用いたバイトコード１４４のコンパイルに伴うコンパイルオーバヘッドは、このコンパイルオーバヘッドが通常予め定められる最大許容レベルを超えないように監視される。こ

のコンパイルオーバヘッドは、実行に対してコンパイルに必要なプロセッサ時間のパーセンテージを用いて表現されることが多い。予め定められた最大オーバヘッドレベルは、特定のシステムの必要性および特性に応じて広範囲に変化しうる。例えば、高い実行性能を与えるように意図された仮想マシンでは、予め定められた最大レベルは、全システム資源の約１０パーセント使用から約３０パーセント使用までの範囲内とすることができる。バイトコード１４４のコンパイルに伴うオーバヘッドがこの所定最大レベルを超える場合、コンパイルオーバヘッドがもっと低ければコンパイルされていたであろうメソッドを実行リスト中に配置する、すなわち待ち行列に入れることができる。この実行リストは、本質的に、コンパイルオーバヘッドがより低いときに処理できる複数のメソッドからなる待ちリストである。すなわち、コンパイルオーバヘッドがより低いときは、実行リスト中のメソッドをコンパイルすることができる。具体的には、コンパイルオーバヘッドとコンピュータシステム１４６に伴う全計算オーバヘッドとの双方が低レベルである時に実行リストを処理することができる。

【００１８】バイトコードが、プログラムの全体的実行における別のアイドル期間（例えば、バイトコードの処理を通じて生じるアイドル期間）中にコンパイルされる場合は、プログラムを監視して休止が生じる時点を判断してもよい。次に図２を参照すると、プログラム、すなわちアプリケーションを実行するプロセスは、休止中にコードをコンパイルする機能を含んでいる。このプロセスを本発明の一実施形態に従って説明する。プログラムの実行はステップ２１０から始まり、ステップ２１２では、プログラムの実行がアイドル期間に到達したかどうかに関して判断がなされる。

【００１９】アイドル期間は、一般に、プログラムが本質的にインアクティブとなるプログラム処理中の休止、例えば思考休止（think pause）である。プログラムは、実行をアクティブに継続する前に、関連オペレーションシステムからタイマ信号などのアクションを待ってもよいし、あるいはユーザ入力を待ってもよい。一般に、思考休止は、プログラムがユーザからのアクションを待っている間に生じる。アイドル期間の長さは、広範囲に変化しうる。例えば、アイドル期間の長さは、わずか数ミリ秒から約２分の１秒まで変化することがある。プログラムがユーザからの入力を待っているためにアイドル期間が生じる場合は、一般にアイドル期間が極めて長くなることがあり、例えば、数秒、数分、数日あるいはそれ以上のオーダとなることがある。

【００２０】一般に、「有用」な休止を認識するためにしきい値が用いられる。例えば、実際の休止が所定のミリ秒数にわたって続くと、システムはその休止を、もう少しのあいだ継続する可能性が高い有用な休止として認識することができる。休止が続きそうな場合、その休止を活用するためにコンパイルを継続できるようにしてもよい。このため、プログラムの実行がアイドル期間に到達したかどうかに関するステップ２１２での判断には、低アクティビティの期間がしきい値（例えば、約１００ミリ秒や約２分の１秒）に到達したかどうかの判断が含まれる。

【００２１】一般に、アイドル期間、すなわち比較的アクティビティの低い期間の発生は、プログラムの実行に伴う中央処理装置（ＣＰＵ）の使用率を監視することによって識別することができる。この他に、全コンピュータシステムに伴うスレッドの状態を監視することによってアイドル期間の存在を判断することもできる。

【００２２】相対的インアクティビティ（relative inactivity）の期間を識別するためにＣＰＵ使用率が監視される場合、ＣＰＵ使用率は、本質的には、ＣＰＵ使用率が特定の使用率しきい値を下回る時点を判断するために監視される。使用率しきい値、すなわち「低アクティビティしきい値」は、広範囲に変化しうる。例えば、使用率しきい値は、オーバヘッドの低いアクティビティ（例えば、カーソルを周期的に明滅するように設定すること）しか実質的に動作していないときにプログラムがアイドルであるとみなされるように設定することができる。ある実施形態では、プログラムの処理中のＣＰＵ使用率が全システム資源の約２０パーセントという使用率しきい値を下回ると、プログラムの全体処理が低アクティビティ期間にあるとみなされる。このような時点では、システム資源の少なくとも約８０パーセントが使用可能な場合がある。

【００２３】当業者であれば理解できるように、スレッドの状態の監視には、スレッドスケジューラの検討（studying）が含まれることがある。全てのスレッドがブロックされたこと、すなわち実質的にどのスレッドも「実行可能（run-able）」な状態にないことをスレッドスケジューラが示す場合は、プログラムがＣＰＵ時間を消費していないことが示唆されている。一般に、ＣＰＵ時間は、スレッドが非ブロック状態となりうるどの時点においても、信号が到着するまでは消費されない。ＣＰＵ時間が消費されていない場合、プログラムは一般にアイドル期間にある。

【００２４】プログラムの実行がアイドル期間にないとステップ２１２で判断されると、プロセスフローはステップ２１０に戻り、ここでプログラムは動作を続ける。この代わりに、アイドル期間があるとステップ２１２で判断されると、コンパイルなどの保留タスク（pending task）が存在するかどうかに関してステップ２１４で判断がなされる。すなわち、アイドル期間中に発生することが予定されたタスクが存在するかどうかに関して判断がなされる。このようなタスクには、一般に、コンパイルや不要部分の整理など、多様なタスクが含まれる。このようなタスクには、これらのタスクのシステム内での

相対的な重要度を用いて優先順位を付けてもよい。ここに記載する実施形態では、これらのタスクを保留コンパイルとして説明する。

【００２５】一般に、保留コンパイルは、二つのソースから得ることができる。すなわち、データベースリストと実行リストである。これらは、コンパイル用の「候補」メソッドのリストである。データベースリストを生成する一つの構成には、コンパイル済メソッドを監視するシステム、例えばデータベースがインライン化されるときにデータベースリストを生成する構成が含まれていてもよい。このようなシステムでは、データベースリストは本質的に、プログラムの前の実行中にコンパイルされたメソッドであって、プログラムの後の実行時にコンパイルされる可能性が高いメソッドと考えられるメソッドからなる「ワーキングセット」、あるいは「候補セット」である。他の機構を用いてデータベースリストを生成したり追加してもよい。一部の実施形態では、データベースリストが、不要部分の整理機能など、他のタスクを含んでいてもよい。

【００２６】上述のように、実行リストは、プログラムの現在の実行中に生成され、複数のメソッドからなる待ち行列、すなわちメソッドの候補セットである。この実行リストは、メソッドのコンパイルを可能にするにはコンパイルオーバヘッドが高すぎると以前に考えられていた事実によってコンパイルから抑制されていたメソッドを含む、すなわち識別してもよい。この実行リストは、解釈済メソッドに関連付けられた呼出しカウンタ、すなわちメソッドが解釈された回数に追随するカウンタ、を検討する独立スィーパプロセス（separate sweeper process）によってリストに定期的に追加されたメソッドを識別していてもよい。一般に、メソッド用の呼出しカウンタは、メソッドが呼び出されるたびに増分される。スィーパは、呼出しカウンタを定期的に検討することにより、最近呼び出されたことのないメソッドの呼出しカウンタが、メソッドをコンパイルに適切とみなすために現在考えられているレベルにあるかどうかを判断することができる。しきい値（例えば、メソッドをコンパイル用と考えるために一般的に到達しなければならないメソッドの呼出し回数）はプログラム実行中に変化しうるので、現在のしきい値を上回る呼出しカウンタを有するメソッドをスィーパによって実行リストに加えてもよい。他の機構を用いて実行リストを生成または追加してもよい。一部の実施形態では、実行リストが、他の保留タスク（例えば、不要部分の整理）に加えて保留コンパイルを含む一般ワークリストに付随していてもよい。

【００２７】ステップ２１４において実行リストおよびデータベースリストの検討を通じて保留タスクがないと判断されると、プロセスフローはステップ２１０に戻り、そこでプログラムは動作を継続する。一方、保留コンパイルがあると判断された場合は、ステップ２１６においてコンパイル等のタスクが実行される。保留コンパイルの実行に伴うステップは、図３を参照しながら以下で説明する。保留コンパイルが実行された後、プロセスフローはステップ２１０に戻り、そこでプログラムは動作を継続する。

【００２８】前述したように、ここに記載する実施形態では、メソッドのコンパイルおよびコンパイル済メソッドの呼出しがメソッドの解釈よりも効率的である可能性が高いと予想される場合には、メソッドがコンパイルされる。過度のコンパイルオーバヘッドを避けるため、このようなコンパイルは、プログラム全体の実行中にアイドル期間が存在するようになるまで遅延させてもよい。図３は、本発明の一実施形態に従ったメソッドに対する遅延コンパイルの実行（すなわち、図２のステップ２１６）に伴うステップを示すプロセスフロー図である。このコンパイル実行プロセスは、ステップ３０２から開始する。ステップ３０２では、最も高い優先順位を持つ候補メソッドが識別される。ある実施形態では、最高優先順位メソッドが、実行リスト内に存在する、すなわち実行リストによって識別される最高の呼出し回数を有するメソッドであってもよい。この他に、最高優先順位メソッドは、実行リスト上の最短予想コンパイル時間を有するメソッドであってもよい。最高優先順位メソッドの識別は、図４を参照しながら後で詳細に説明する。

【００２９】ステップ３０２で最高優先順位メソッドが識別された後、ステップ３０４では、最高優先順位メソッドが既にコンパイルされているかどうかに関して判断がなされる。最高優先順位メソッドは、種々の理由によって既にコンパイルされている可能性がある。例えば、メソッドが実行リスト内に配置された後、全体的なプログラム実行の過程において、コンパイルオーバヘッドが低い間にメソッドが呼出されコンパイルされる場合がある。このため、このメソッドは既にコンパイルされており、また、実行リストを繰り返し更新するのは時間がかかることから、このメソッドが実行リストから除去されていない可能性もある。最高優先順位メソッドも、これがデータベースリストから得られる場合は、以前にコンパイルされている可能性がある。すなわち、実行リストとデータベースリストの双方に同じメソッドが現れる場合は、例えばメソッドが実行リストを用いて既にコンパイルされており、データベースリストが損失が大きいために更新されていないと、そのメソッドはデータベースリスト中の最高優先順位メソッドとして識別されるときに既にコンパイルされている可能性がある。

【００３０】ステップ３０４での判断が、最高優先順位メソッドが既にコンパイルされているというものである場合、このメソッドは適切なリスト、すなわち実行リストかデータベースリストのいずれかから取り除くことができ、プロセスフローはステップ３０２に戻り、ここで新たな最高優先順位メソッドが識別される。この代わり

に、最高優先順位メソッドが前にコンパイルされていなかったと判断された場合は、ステップ３０６において最高優先順位メソッドのコンパイルが開始する。すなわち、最高優先順位メソッドのコンパイルが初期化される。

【００３１】最高優先順位メソッドのコンパイル中、割込みが受信される場合がある。一般に、割込みには、タイマ信号およびユーザ入力（例えばキーボード入力）が含まれことがある。但し、これに限定されるものではない。ステップ３０８では、割込みを受け取ったかどうかが判断される。当業者であれば分かるように、割込みが受信されると、必要に応じて割込みを処理するために、最高優先順位メソッドのコンパイルが自動的に停止することがある。

【００３２】割込みが受信されていた場合、ステップ３１４において最高優先順位メソッドのコンパイルが所定の時間にわたって継続することが許容される。ある実施形態では、この所定時間は、本質的には、少なくとも一つのスレッドを実行可能状態にセットする信号が受信された後におけるコンパイルの「生存時間」である。割込みが受信された後にコンパイルが短時間継続できるようにすることで、コンパイルの完了が近い場合に、コンパイルを完了させる機会が与えられる。

【００３３】この所定時間は広範に変化しうるが、この所定時間は一般的には数ミリ秒、例えば「Ｘミリ秒」である。約１０ミリ秒から約２００ミリ秒の範囲内で、より短い時間が好ましい。というのも、ユーザ入力に応答した約２００ミリ秒よりも長い時間の遅延は知覚することができ、従ってユーザをいらだたせることが多いことが分かったからである。この時間は、実行可能状態にあるスレッドの優先順位または信号のソースに基づいて変化する可能性がある。ここで、より高い優先順位の信号またはスレッドには、より短い時間が使用される。

【００３４】ステップ３１４において所定時間にわたる継続がコンパイルに許された後、ステップ３１６では、この所定時間の間にコンパイルが成功して完了したかどうかに関して判断がなされる。コンパイルが完了している場合、プロセスフローは図２のステップ２１０に移動し、ここでプログラムの全体的な実行が継続する。コンパイルが完了すると、新たにコンパイルされたメソッドも、そのメソッドがどのリストから得られたかに応じて実行リストまたはデータベースリストのいずれかから取り除かれることになる。

【００３５】所定時間の間にコンパイルが完了していない場合、プロセスフローはステップ３１６からステップ３１８に移動し、ここでコンパイルは打ち切られる。コンパイルの完了を許すと全プログラムの実行や特に割込みの処理に大きな遅延を生じることがあるという事実により、コンパイルの完了は許されない。コンパイルが打ち切られると、「クリーンアップ」が生じ、例えば、当業者であれば分かるように、試みたコンパイルで使用されたシステム資源が放棄される。コンパイルが打ち切られると、プロセスフローは図２のステップ２１０に移動し、ここでプログラムの全体的実行が続く。

【００３６】ここで、ステップ３０８に戻る。最高優先順位メソッドのコンパイル中に何ら割込みが受信されなかった場合には、ステップ３１０でコンパイルが継続する。ステップ３１２では、コンパイルが完了したかどうかに関して判断がなされる。コンパイルが完了したと判断される場合、最高優先順位メソッドを適切なリストから取り除くことができ、プロセスフローはステップ３０２に戻り、ここで新しい最高優先順位メソッドが識別される。この他に、コンパイルが完了していないとステップ３１２で判断された場合は、コンパイルが完了するか、あるいはステップ３０８における割込みの受信によってコンパイルが打ち切られるまで、コンパイルは継続することを許される。

【００３７】一般に、実行リストおよびデータベースリスト中のメソッドがコンパイルされる「順序」は、多様な要因に基づいて決定することができる。各リスト中のメソッドには、同じリスト中の他のメソッドに対する優先順位の値を割り当てることができる。優先順位の値は、一般に、プログラムの全体実行中における任意の時点に割り当てることができ、更にプログラム実行の過程を通じて更新することも可能である。例えば、リスト中の優先順位値は、メソッドがリストに追加されるたびに計算および更新、すなわち再評価することができる。実行リストおよびデータベースリスト中のメソッドの優先順位がコンパイル用にメソッドが選択される時に最新となるように、優先順位の値を各アイドル期間の開始時に再評価してもよい。

【００３８】メソッドの優先順位の値は、複数の要因の任意の組合せに基づいて計算することができる。このような要因には、メソッドの呼出し回数、リスト内におけるメソッドの位置、メソッドの推定コンパイル時間、およびプログラムの全体的実行中における現行アイドル期間の長さが含まれる。但し、これらに限定されるものではない。この他に、ある実施形態では、メソッドの優先順位値を本質的にランダムに割り当てることができる。

【００３９】リスト中に最高の呼出し回数を有するメソッドは、最高のコンパイル優先順位を有する可能性がある。というのも、このメソッドは、本質的に最も高い頻度で呼び出された解釈済メソッドだからである。リスト内におけるメソッドの位置も、メソッドに割り当てられた優先順位値に影響を与える可能性がある。これは、リストに最も新しく追加されたメソッドは、多くの場合、近い将来に必要となる可能性が高いからである。リスト中の最短推定コンパイル時間を有するメソッドは、最高のコンパイル優先順位を有する可能性がある。これは、より短い推定コンパイル時間を有するメソッドのコンパ

イルは、一般に、より長い推定コンパイル時間を有するメソッドのコンパイルよりもアイドル期間中に完了する可能性が高いからである。一般に、メソッドのコンパイル時間は、メソッドの長さによって推定することができる。例えば、比較的短いメソッドは、通常、比較的短いコンパイル時間を有することになる。

【００４０】現行アイドル期間の長さも、メソッドの優先順位値に影響を与える可能性がある。例えば、現行アイドル期間が既に比較的長い場合、アイドル期間がより長く続く可能性は一般に高い。長いアイドル期間は、しばしばプログラムの全体的実行が長時間にわたって生じているコンピュータをユーザが離れる結果となることがある。このように、現行のアイドル期間が既に比較的長いと判断される場合にはアイドル期間が更に長く続く可能性が一般的にかなり高いことが分かった。従って、アイドル期間が比較的長いと予想される場合、より長い予測コンパイル時間を有するメソッドは、より高いコンパイル優先順位を有していてもよい。というのも、より短い予測コンパイル時間を有するメソッドは、通常、後続の、より短いアイドル期間中にすぐにコンパイルすることができるからである。

【００４１】前述のように、メソッドの優先順位値は、多様な要因に基づくことがある。換言すると、優先順位関数を用いて優先順位値を決定することができる。このような優先順位関数の形は、一般に広範囲に変化しうる。例えば、優先順位関数は、あるメソッドについて優先順位値を得るために、そのメソッドについての呼出し回数をそのメソッドについての推定コンパイル時間で除算するものであってもよい。この他に、優先順位関数は、短い推定コンパイル時間を有するメソッドについての呼出し回数の使用や、長い推定コンパイル時間を有するメソッドに対する定数で呼出し回数を除算したものの使用を含んでいてもよい。

【００４２】次に図４を参照しながら、最高の優先順位値を有するメソッドの識別（すなわち、図３のステップ３０２）に伴うステップを本発明の一実施形態に従って説明する。最高の優先順位を有するメソッドの識別は、プログラムに付随する実行リスト中にメソッドが存在するかどうかの判断を伴うステップ４０２から開始する。

【００４３】一般に、実行リスト中のメソッドは、データベースリスト中のメソッドよりも高いコンパイル優先順位を有しているものと考えられる。実行リスト中のメソッドは、より効率的に実行することによって付随コンパイルオーバヘッドを克服する可能性が高いものとしてプログラム実行の過程中に識別されたメソッドである。というのも、これらのメソッドは繰り返し呼び出されるからである。この他に、データベースリスト中のメソッドは、潜在的に繰り返し呼び出される可能性のあるメソッドであり、従って、これらのメソッドのコンパイルが有益である可能性は高い。言い換えると、実行リストは、現在の、すなわち実際のメソッド使用率を反映しており、データベースリストは、本質的にメソッドの長期使用率を反映している。従って、実行リストからのメソッドは、通常、データベースリストからのメソッドの前にコンパイル用に選択される。これは、実行リスト中のメソッドをコンパイルする方が、これらのメソッドを使用するプログラムの全体的実行を効率的にする可能性が高いからである。

【００４４】ステップ４０２での判断が実行リスト中にメソッドが存在するというものである場合、ステップ４０４において、実行リスト中で最も高い優先順位を有するメソッドがコンパイルのために選択される。実行リストからメソッドが選択されると、最高優先順位メソッドを識別するステップは完了する。この他に、ステップ４０２での判断が実行リスト中にメソッドが存在しないというものである場合、データベースリスト中の最高優先順位メソッドがコンパイル用に選択される。データベースリストからメソッドが選択されると、最高優先順位メソッドを識別するプロセスは完了する。

【００４５】本発明は、一般に、適切なコンピュータシステム上で実施することができる。特に、プログラムの全体的実行中のアイドル期間における待機メソッド（queuedmethod）のコンパイルは、適切な仮想マシン（例えば、図６を参照して後述する仮想マシン）を用いて達成することができる。図５は、本発明の実施に適した代表的な汎用コンピュータシステムを示している。このコンピュータシステム５３０は、複数のメモリ素子に結合された任意の数のプロセッサ５３２（これは、中央処理装置、すなわちＣＰＵとも呼ばれる）を含んでいる。これらのメモリ素子には、一次記憶装置５３４（通常は、リードオンリメモリ、すなわちＲＯＭ）と一次記憶装置５３６（通常は、ランダムアクセスメモリ、すなわちＲＡＭ）とが含まれている。

【００４６】当業者であれば分かるように、コンピュータシステム５３０、またはより具体的にＣＰＵ５３２は、仮想マシンをサポートするように構成されている。コンピュータシステム５３０でサポートされる仮想マシンの一例は、図６を参照しながら後述する。この技術分野では周知のように、ＲＯＭは、データおよび命令を単方向でＣＰＵ５３２に転送するように作用するが、ＲＡＭは、通常、データおよび命令を双方向的に転送するために使用される。一般に、ＣＰＵ５３２は、任意の数のプロセッサを含んでいてもよい。一次記憶装置５３４、５３６は、双方とも任意の適切なコンピュータ読取り可能媒体を含んでいてもよい。ＣＰＵ５３２には二次記憶媒体５３８（通常は、マスメモリ素子）も双方向に結合されており、付加的なデータ記憶容量を与えている。マスメモリ素子５３８は、コンピュータコード、データなどを含むプログラムを記憶するために使用可能なコンピュータ読取り可能媒体である。通常、マスメモリ素子５

３８は、一次記憶装置５３４、５３６よりも一般的に遅いハードディスクやテープなどの記憶媒体である。マスメモリ記憶装置９３８は、磁気テープリーダや紙テープリーダ、あるいは他の周知装置の形をとることができる。マスメモリ素子５３８内に保持される情報は、適切な場合に、仮想記憶としてＲＡＭ５３４の一部として標準的な方式で組み込まれる。特定の一次記憶装置５３６（ＣＤ－ＲＯＭなど）も、ＣＰＵ５３２に単方向でデータを受け渡すことができる。

【００４７】ＣＰＵ５３２は、一つ以上の入出力装置５４０にも結合されている。この入出力装置としては、ビデオモニタ、トラックボール、マウス、キーボード、マイクロフォン、タッチセンシティブディスプレイ、トランスジューサカードリーダ、磁気テープまたは紙テープリーダ、タブレット、スタイラス、音声または手書き認識装置、あるいは他の周知の入力装置（例えば、当然のことではあるが、他のコンピュータ）を挙げることができる。但し、これらに限定されるわけではない。最後に、ＣＰＵ５３２は、オプションとして、符号５１２で全体的に示されるネットワーク接続を用いてコンピュータまたは通信ネットワーク（例えば、ローカルエリアネットワーク、インターネットネットワーク、イントラネットネットワーク）に結合されていてもよい。このようなネットワーク接続を用いると、ＣＰＵ５３２は、上述の方法ステップを実行する過程でネットワークから情報を受け取ったり、ネットワークに情報を出力するとができるものと考えられる。このような情報は、ＣＰＵ５３２を用いて実行すべき一連の命令として表されることが多いが、例えば搬送波に具体化されるコンピュータデータ信号の形でネットワークから受け取ったりネットワークに出力することができる。上述の装置およびマテリアルは、コンピュータハードウェアおよびソフトウェア技術の当業者には良く知られたものであろう。

【００４８】前述のように、仮想マシンはコンピュータシステム５３０上で動作することができる。図６は、図５のコンピュータシステム５３０によってサポートされ、本発明を実施するのに適した仮想マシンの線図である。コンピュータプログラム、例えばカリフォルニア州マウンテンビューのサンマイクロシステムズ（Sun Microsystems）によって開発されたJava（商標）プログラミング言語で書かれたコンピュータプログラムが実行されると、翻訳時環境６０５内のコンパイラ６２０にソースコード６１０が提供される。コンパイラ６２０は、ソースコード６１０をバイトコード６３０に翻訳する。一般に、ソースコード６１０は、ソースコード６１０がソフトウェア開発者によって作成される時にバイトコード６３０に翻訳される。

【００４９】一般に、バイトコード６３０は、ネットワーク（例えば、図５のネットワーク５１２）を通じて複写、ダウンロード、または配布したり、図５の一次記憶装置５３４などの記憶装置に記憶することができる。ここに記載する実施形態では、バイトコード６３０はプラットフォームに依存しない。すなわち、バイトコード６３０は、適切な仮想マシン６４０を動作させている実質的に任意のコンピュータシステム上で実行することができる。例えば、Java（商標）環境では、Java（商標）仮想マシンを動作させているコンピュータシステム上でバイトコード６３０を実行することができる。

【００５０】バイトコード６３０は、仮想マシン６４０を含む実行時環境６３５に提供される。実行時環境６３５は、一般に、図５のＣＰＵ５３２などのプロセッサを用いて実行することができる。仮想マシン６４０は、コンパイラ６４２、インタープリタ６４４、およびランタイムシステム６４６を含んでいる。バイトコード６３０は、一般に、コンパイラ６４２またはインタープリタ６４４のいずれかに提供することができる。

【００５１】バイトコード６３０がコンパイラ６４２に提供される場合は、上述のように、バイトコード６３０に含まれるメソッドが機械語命令にコンパイルされる。一方、バイトコード６３０がインタープリタ６４４に提供される場合は、バイトコード６３０が一度に１バイトコードずつインタープリタ６４４に読み込まれる。次いで、インタープリタ６４４は、各バイトコードがインタープリタ６４４に読み込まれるのに伴って、各バイトコードによって定義される操作を実行する。一般に、インタープリタ６４４は、バイトコード６３０を処理して、バイトコード６３０に関連付けられた操作をほぼ連続的に実行する。

【００５２】メソッドがオペレーティングシステム６６０から呼び出されると、そのメソッドを解釈済メソッドとして呼び出すべきであると判断される場合は、ランタイムシステム６４６はそのメソッドをインタープリタ６４４から取得することができる。一方、メソッドをコンパイル済メソッドとして呼び出すべきであると判断される場合は、ランタイムシステム６４６はコンパイラ６４２を起動する。この後、コンパイラ６４２は、バイトコード６３０から機械語命令を生成し、その機械語命令を実行する。一般に、この機械語命令は、仮想マシン６４０が終了するときに廃棄される。

【００５３】本発明の実施形態を数個しか記載しなかったが、本発明は、本発明の趣旨または範囲から逸脱することなく他の多くの特定形態に具現化することができる。例えば、アイドル期間中のコンパイル実行に伴うステップは、順序を変えたり、削除または追加することができる。一般に、本発明の方法に伴うステップは、本発明の趣旨または範囲から逸脱することなく順序を変えたり、取り除いたり、追加することができる。

【００５４】上記では、コンパイル用の最高優先順位メソッドを、実行リストおよびデータベースリスト中の実質的に全てのメソッドから選択されたものとして述べて

きた。実行リストまたはデータベースリスト、あるいはこの双方に含まれる全メソッドを探索して最高のコンパイル優先順位を持つメソッドの位置を特定する方法では、実行リストが膨大な場合に損失が大きいことがわかる。従って、ある実施形態では、実行リストのほぼ全体を探索する代わりに、実行リストの部分集合を探索することで、コンパイル用の最高優先順位メソッドを識別することができる。例えば、実行リストから最初の「Ｎ」個の候補メソッドを探索することにより、この最初のＮ個の候補メソッドの中から最高優先順位メソッドを識別することができる。

【００５５】前述のように、メソッドのコンパイル優先順位は、呼出しカウンタを用いることで少なくとも一部は識別することができる。呼出しカウンタは、一般に、メソッドがアクセスされるたびに増分されるカウンタである。一部の実施形態では、呼出しカウンタが時間の経過とともに減衰してもよい。例えば、メソッドが繰り返し呼び出され、プログラム実行の開始時にコンパイルが予定されていたが、再び呼び出されない場合、呼出しカウンタを減衰させてメソッドのコンパイル優先順位を低減してもよい。メソッド用の呼出しカウンタが減衰、例えば指数関数的に減衰してしまうと、メソッドのコンパイルがもはや望ましくない場合がある。このため、このようなメソッドを定期的に実行リストから取り除いてもよい。

【００５６】同様に、コンパイルの実行の進行中、コンパイルを予定されているメソッドが前にコンパイルされたものであるかどうかに関して判断がなされる。この他に、既にコンパイルされたメソッドを、実行リストおよびデータベースリストの一方または双方から適宜取り除くことができる。一般に、コンパイル済メソッドについての実行リストおよびデータベースリストの検査に伴うオーバヘッドは、比較的大きいことがある。しかしながら、コンパイル済メソッドは、本発明の趣旨または範囲から逸脱することなく、定期的に（例えば、アイドル期間の開始時に）取り除くことができる。

【００５７】これまで実行リストは、コンパイルされるべき複数の解釈済メソッドからなるリスト、すなわち待ち行列として一般的に説明してきたが、実行リストは、アイドル期間が存在するようになるまで再コンパイルから遅延させられたコンパイル済メソッドを含んでいてもよい。例えば、二つのレベルのコンパイラを有するシステムでは、いったんメソッドがコンパイルされてしまうと、そのメソッドが繰り返し実行される場合は、そのメソッドをより効率良く再コンパイルすることが望ましいことが分かる。このようなシステムでは、再コンパイル、すなわち二次コンパイルを予定されているメソッドが、コンパイルを予定された解釈済メソッドとして同じ実行リスト中に含まれていてもよい。しかしながら一方で、再コンパイルが予定されたメソッドが別個の実行リストに含まれていてもよい。

【００５８】ある実施形態では、実行リストは、一般に、保留コンパイルに加えて、または保留コンパイルの代わりに遅延タスクを含むワークリストであってもよい。このようなタスクに優先順位を付けてもよい。ワークリストからのタスクが動作している場合は、割込みが受信されると、そのタスクを完了できるようになっていてもよい。例えば、タスクが不要部分の整理を含む場合、不要部分整理プロセスを中断し、不要部分整理プロセスの「バックアウト」をすることは、このプロセスを完了する以上に犠牲が大きいことがある。このため、不要部分の整理を完了することが許される場合がある。一般に、タスクも、割込みが受信されたときには中断することができる。従って、本例は例示であって限定的なものではないと考えるべきであり、本発明は、本明細書で挙げた詳細に限定されるべきではなく、均等物の全範囲に沿って特許請求の範囲内で変形を加えることができる。

【図面の簡単な説明】

【図１】本発明の一実施形態に従ってコードを動的コンパイルするコンピュータシステムのブロック図である。

【図２】本発明の一実施形態に従って休止中にコードをコンパイルする機能を有するプログラムの実行に伴うステップを示すプロセスフローチャートである。

【図３】本発明の一実施形態に係るメソッドへのコンパイルの実行、すなわち図２のステップ２１６、に伴うステップを示すプロセスフローチャートである。

【図４】本発明の一実施形態に係る最高優先順位メソッドの識別、すなわち図３のステップ３０２に伴うステップを示すプロセスフローチャートである。

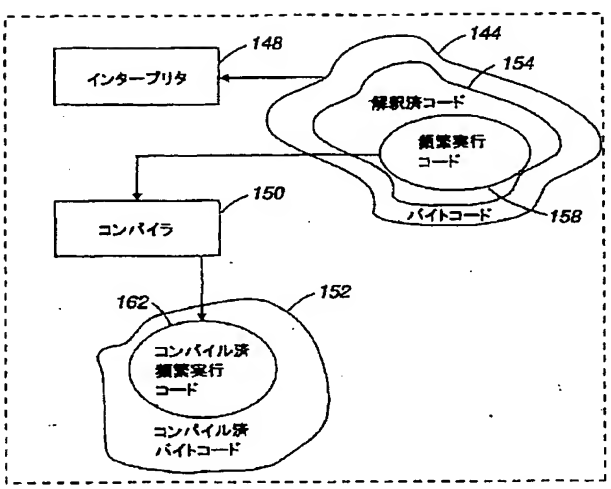
【図５】本発明を実施するのに適した典型的な汎用コンピュータシステムを示す図である。

【図６】本発明を実施するのに適した仮想マシンの線図である。

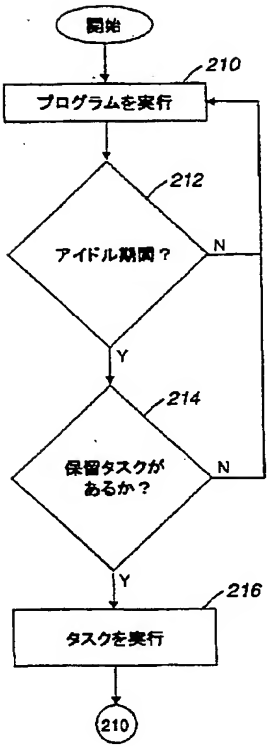
【符号の説明】

１４４…バイトコード、１４８…インタープリタ、１５０…コンパイラ、１５２…コンパイル済バイトコード、１５４…解釈済バイトコード、１５８…頻繁実行コード、１６２…コンパイル済頻繁実行コード。

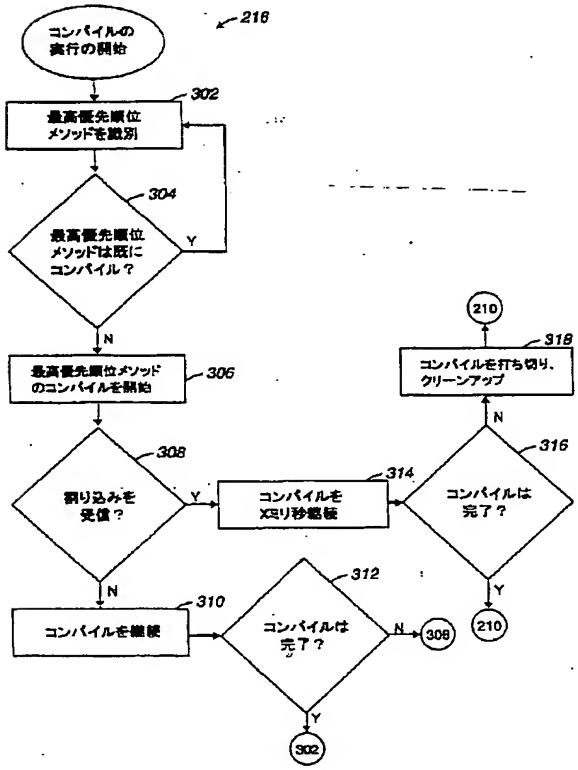
【図１】


148
インタープリタ
144
154
解釈済コード
頻繁実行コード
バイトコード
158
150
コンパイラ
162
152
コンパイル済頻繁実行コード
コンパイル済バイトコード

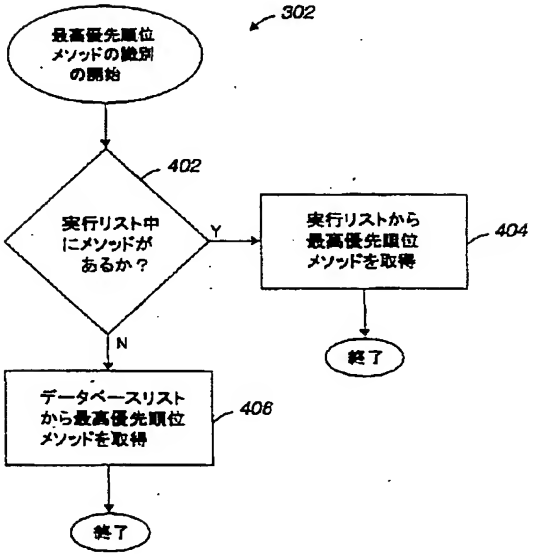
【図２】


開始
210
プログラムを実行
212
アイドル期間？
N
Y
214
保留タスクがあるか？
N
Y
216
タスクを実行
210

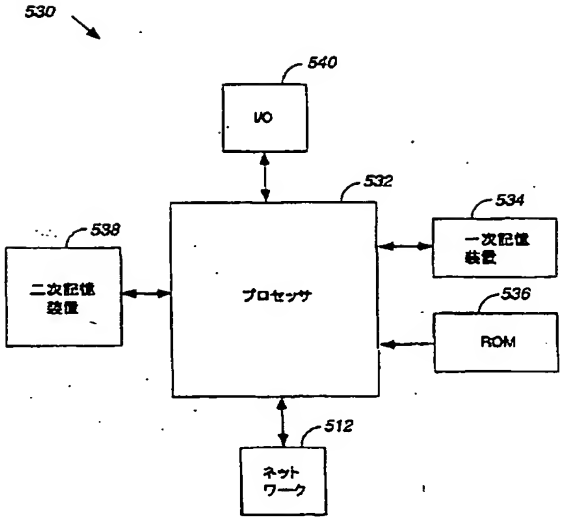
【図３】


コンパイルの実行の開始
216
302
最高優先順位メソッドを識別
304
最高優先順位メソッドは既にコンパイル？
Y
N
306
最高優先順位メソッドのコンパイルを開始
308
割り込みを受信？
Y
314
コンパイルをXミリ秒継続
316
コンパイルは完了？
N
318
コンパイルを打ち切り、クリーンアップ
210
Y
210
N
310
コンパイルを継続
312
コンパイルは完了？
N
306
Y
302

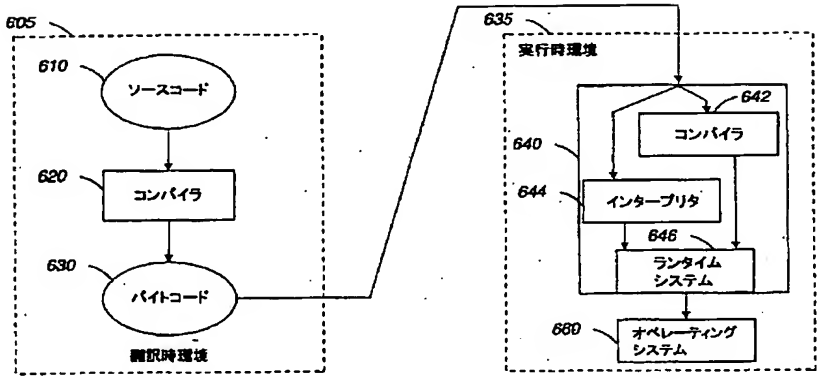
【図４】


302
最高優先順位メソッドの識別の開始
402
実行リスト中にメソッドがあるか？
Y
404
実行リストから最高優先順位メソッドを取得
終了
N
406
データベースリストから最高優先順位メソッドを取得
終了

【図５】



【図６】



フロントページの続き

(72)発明者 ラーズ バク
アメリカ合衆国, カリフォルニア州,
パロ アルト, コリナ ウェイ 3782

〔外国語明細書〕

1. Title of Invention

METHOD AND APPARATUS FOR PERFORMING BYTE-CODE OPTIMIZATION DURING PAUSES

2. Claims

What is claimed is:

1. A computer-implemented method for dynamically compiling methods during an idle period in the processing of a computer program, the computer-implemented method comprising:

identifying the idle period;

identifying a first method selected from a plurality of methods during the idle period, the plurality of methods being included in the computer program, wherein the computer program is arranged to execute both interpreted program code and compiled program code; and

initializing a compilation of the first method, wherein the initialization of the compilation of the first method occurs during the idle period.

2. A computer-implemented method as recited in claim 1 wherein the plurality of methods is referenced in a first list associated with the computer program and the first method is a highest priority method associated with the first list.

3. A computer-implemented method as recited in claim 1 wherein a first set of methods selected from the plurality of methods is associated with a first list and a second set of methods selected from the plurality of methods is associated with a second list, the first list being created during the processing of the computer program the second list being created before the processing of the computer program, wherein identifying the first method includes:

selecting the first method as a highest priority method from the first list.

4. A computer-implemented method as recited in claim 1 wherein a first set of methods selected from the plurality of methods is associated with a first list and a second set of methods selected from the plurality of methods is associated with a second list, the first list being created during the processing of the computer program the second list being created before the processing of the computer program, wherein identifying the first method includes:

selecting the first method as a highest priority method from the second list.

5. A computer-implemented method as recited in any one of the preceding claims further including:

determining whether an interrupt is received by the computer program after the initialization of the compilation of the first method;

continuing the compilation of the first method for a predetermined period of time when it is determined that an interrupt has been received;

determining whether the compilation of the first method is completed after the predetermined period of time; and

aborting the compilation of the first method when it is determined that the compilation of the first method is not completed after the predetermined period of time.

6. A computer-implemented method for dynamically compiling methods during a period of low activity in the overall processing of a computer program, the computer-implemented method comprising:

identifying the period of low activity;

identifying a first method selected from a plurality of methods during the period of low activity, the plurality of methods being included in the computer program;

determining whether the first method is compiled, wherein the determination is made during the period of low activity; and

initializing a compilation of the first method during the period of low activity when it is determined that the first method is not compiled.

7. A computer-implemented method as recited in claim 6 further including:

identifying a second method selected from the plurality of methods during the period of low activity when it is determined that the first method is compiled;

determining whether the second method is compiled, wherein the determination of whether the second method is compiled occurs during the period of low activity; and

initializing a compilation of the second method during the period of low activity when it is determined that the second method is not compiled.

8. A computer-implemented method as recited in one of claims 6 and 7 further including:

determining a priority value for each of the plurality of methods, wherein the first method is identified based on the priority value for the first method.

9. A computer-implemented method as recited in claim 8 wherein determining the priority value for each of the plurality of methods includes processing an

invocation counter associated with each of the plurality of methods during the period of low activity.

10. A computer-implemented method as recited in claim 8 wherein determining the priority value for each of the plurality of methods includes estimating a compilation time for each of the plurality of methods during the period of low activity.

11. A computer-implemented method as recited in claim 6 wherein a first set of methods selected from the plurality of methods is associated with a first list and a second set of methods selected from the plurality of methods is associated with a second list, the first list being created during the processing of the computer program the second list being created before the processing of the computer program, wherein identifying the first method includes:

selecting the first method as a highest priority method from the first list.

12. A computer-implemented method as recited in any one of claims 6-11 further including:

determining whether a interrupt is received by the computer program after the initialization of the compilation of the first method;

continuing the compilation of the first method for a predetermined period of time when it is determined that an interrupt has been received;

determining whether the compilation of the first method is completed after the predetermined period of time; and

aborting the compilation of the first method when it is determined that the compilation of the first method is not completed after the predetermined period of time.

13. A computer-implemented method as recited in claim 12 further including:

completing the compilation of the first method when it is determined that an interrupt has not been received;

identifying a second method selected from a plurality of methods during the period of low activity;

determining whether the second method is compiled during the period of low activity; and

initializing a compilation of the second method during the period of low activity when it is determined that the second method is not compiled.

14. A computer-implemented method for dynamically processing delayed tasks during an idle period in the processing of a computer program, the computer-implemented method comprising:

identifying the idle period in the processing of the computer program;

identifying a first task selected from a plurality of tasks during the idle period in the processing of the computer program, the plurality of tasks being included in the computer program, wherein the computer program is arranged to execute both interpreted program code and compiled program code; and

starting the first task, wherein the starting of the first task occurs during the idle period.

15. A computer-implemented method as recited in claim 14 further including completing the first task when an interrupt is received.

16. A computer-implemented method as recited in claim 14 further including suspending the first task when an interrupt is received.

17. A computer system for dynamically compiling byte-coded methods during a period of low activity in the processing of a computer program, the methods being associated with the computer program, the computer system comprising:

a mechanism for identifying the period of low activity;

a mechanism for identifying a first method selected from a plurality of methods during the period of low activity; and

a compiler for initializing a compilation of the first method during the period of low activity.

18. A computer system as recited in claim 17 further including:

a first list arranged to reference a first set of the plurality of methods, wherein the first list is associated with the computer program and the first method is a highest priority method associated with the first list; and

a second list arranged to reference a second set of the plurality of methods, wherein the second list is created before the processing of the computer program and the first list is created during the processing of the computer program.

19. A computer system as recited in claim 17 further including:

an interrupt processor arranged to receive an interrupt signal during the compilation of the first method, the interrupt processor being in communication with the compiler, wherein the compiler is further arranged to continue the compilation of the first method for a predetermined period of time in response to a signal from the interrupt processor; and

an aborting mechanism, the aborting mechanism being in communication with the compiler, wherein the aborting mechanism is arranged to abort the compilation of the first method.

20. A computer system as recited in any one of claims 17-19 further including:

a diagnostic mechanism, the diagnostic mechanism being arranged to identify when the first method is compiled.

21. A computer-readable medium including computer program code devices arranged to dynamically compile methods during an idle period in the processing of a computer program, the computer-readable medium comprising:

computer program code devices that identify the idle period;

computer program code devices that identify a first method selected from a plurality of methods during the idle period, the plurality of methods being included in the computer program; and

computer program code devices that initialize a compilation of the first method during the idle period.

22. A computer-readable medium as recited in claim 22 wherein the computer program code devices are embodied in a carrier wave.

3. Detailed description of Invention

BACKGROUND OF THE INVENTION

1. Field of Invention

The present invention relates generally to methods and apparatus for optimizing the execution of software applications. More particularly, the present invention relates to methods and apparatus for performing compilations or other activities during pauses in the overall execution of a computer program to optimize the use of computer system resources.

2. Description of Relevant Art

Computer systems are often linked across a network, *e.g.*, local area networks, intranets and internets, of computer systems such that they may share resources such as software applications. In general, software applications, or computer programs, may be delivered in different formats to different computer systems, due to the fact that each computer system requires software applications to be in a specific format. Alternatively, the computer programs may be delivered to a computer system in a machine-independent form, *i.e.*, as byte codes, in order to enable one form of a computer program to be utilized by many different computer systems.

When computer programs are delivered in a machine-independent form, the programs may be interpreted directly, or the programs may be translated into machine-dependent code, *i.e.*, "machine code." Programs which are interpreted directly occupy less space in a computer system than programs which are translated

into machine code. However, programs which are interpreted directly have slower execution speeds than programs which are translated into machine code, in most cases. As such, the determination of whether or not to interpret a computer program directly, in lieu of translating the computer program into machine code, is often based on the relative importance of space in relation to execution speed.

One approach to compilation is to compile methods within a program that is actively executing when they are first called for execution. This approach is frequently referred to as either dynamic or "runtime" compilation. One problem with dynamic compilation is that the compilation overhead associated with the program may become excessive. That is, when a relatively large number of methods must be compiled at about the same time, the overhead associated with the compilation of the program may be at a level that adversely affects the overall execution of the program. In general, an overuse of system resources for compilation may lead to a less efficient, slower execution of the program. In addition, the time consumed by a compiler which is compiling methods during active program execution may introduce user-visible pauses into the execution of the program. By way of example, when a user inputs a command into the program while the compilation overhead of the program is high, there may be a delay in the response to the user input. Such delays in response may be annoying to a user. Therefore, mechanisms that improve the apparent efficiency of dynamic compilation of methods in a computer program would be desirable.

SUMMARY OF THE INVENTION

To achieve the foregoing and other objects of the invention, methods and apparatus for dynamically compiling methods during idle periods in the execution of a computer program are disclosed. The described methods are particularly suitable for use in computer systems that are arranged to execute both interpreted and compiled byte codes. According to one aspect of the present invention, an idle or low overhead period in the processing of a computer program is identified. One or more methods that have been identified for compilation are then dynamically compiled during the identified idle period. In some embodiments, methods to be dynamically compiled are referenced in one or more lists. The lists may be prioritized to facilitate the compilation of the highest priority methods first. In one embodiment, a pair of compilation lists are provided with a first one of the compilation lists being a database list created prior to processing the computer program while the other being an execution list created during the processing of the computer program.

In one embodiment, if an interrupt is received during the compilation of a method, the compilation is allowed to continue for a predetermined period of time. If the compilation is not completed during the predetermined period of time, then the compilation is aborted.

According to still another aspect of the present invention, a computer system for dynamically compiling byte-coded methods during a period of low activity in the processing of a computer program includes a mechanism for identifying a period of low activity. The system further includes a mechanism for identifying a method for

compilation during the period of low activity. A compiler is arranged to initialize a compilation of the identified method during the period of low activity. In one embodiment, the system also includes an interrupt processor which is arranged to handle interrupt signals received during the compilation of the identified method, and an aborting mechanism which aborts the compilation of the method when necessary.

These and other advantages of the present invention will become apparent upon reading the following detailed descriptions and studying the various figures of the drawings.

The invention may best be understood by reference to the following description taken in conjunction with the accompanying drawings.

DETAILED DESCRIPTION OF THE EMBODIMENTS

As described above, the dynamic compilation of methods associated with a byte-coded program while the program is actively executing may cause the program to execute inefficiently. This inefficiency may be a result of an excessive use of available computer system resources by the compiler, or compilation overhead. The time consumed by a compiler that is dynamically compiling methods during active program execution may introduce delays into the execution of the program. When such delays are perceptible to a user, the delays may be considered to be unacceptable, especially when the length of a delay is relatively long, *e.g.*, more than approximately 200 milliseconds.

One approach to improving the performance of such programs is to permit the mixing of both interpreted and dynamically compiled byte codes. Within such a system, decisions must still be made as to when it is efficient to compile a particular method. In the present invention, method and an apparatus are described for delaying

the dynamic compilation of methods in a computer program when compilation overhead is relatively high. More specifically, when it is determined that compilation overhead is too high, the dynamic compilation of at least some of the methods may be delayed until there are periods of relatively lower activity in the overall processing of the program.

By delaying the dynamic compilation of at least some byte-coded methods which are slated for compilation, computer resources may be more efficiently utilized. While a computer program is essentially inactive during processing, *e.g.*, when the computer program is idle while awaiting input from a user, the overall resource utilization overhead associated with the execution of the program is generally relatively low. During such periods of low activity, available system resources may be used to perform compilations of methods which have previously been identified as being methods which are to be compiled. Performing compilations while the computation overhead of the computer program is low allows for the efficient usage of overall system resources and, typically, does not cause delays which are noticeable to a user.

With initial reference to Figure 1, a computer system which allows byte codes to be dynamically compiled will be described in accordance with an embodiment of the present invention. Byte codes 144 are provided as input to a computer system 146 at run-time. Byte codes 144, which may generally be organized as a computer program, are typically arranged in parseable segments such as methods, or routines. In one embodiment, byte codes 144 may be provided by a compile-time environment

within a virtual machine to a run-time environment, *e.g.*, computer system 146, within the same virtual machine. One suitable virtual machine on which the present invention may be implemented will be discussed below in more detail with respect to Figure 6.

When byte codes 144 are provided to computer system 146, byte codes 144 may be processed with an interpreter 148. Alternatively, byte codes 144 may be compiled by a compiler 150 to produce compiled byte codes 152. It should be appreciated that byte codes 144 may generally be inputted to both interpreter 148 and compiler 150 for processing, substantially simultaneously.

Each time a method is invoked, if the method is not compiled, byte codes 144 associated with the method are interpreted using interpreter 148. In one embodiment, a measure of how many times a method is interpreted is maintained. Such measures may include a counter, *e.g.*, an invocation counter, which is included in each interpreted method, and is incremented each time the method is interpreted.

When the number of times a method is interpreted exceeds a threshold, *i.e.*, a limiting value, the method may be compiled using compiler 150. Repeatedly interpreting a method, which is included in frequently executed code 158, may be inefficient, as interpreted byte code 154 generally executes slower, or less efficiently, than compiled code. Compiling frequently executed code 158 generally may allow methods embodied in frequently executed code 158 to be executed more efficiently,

as time-savings gained by compiling the method is likely to compensate for the compilation overhead associated with the compilation process.

During run-time, the compilation overhead associated with compiling byte codes 144 using compiler 150 is monitored to ensure that the compilation overhead does not exceed a typically predetermined maximum acceptable level. The compilation overhead is often expressed in terms of the percentage of processor time required for compilation as opposed to execution. The predetermined maximum overhead level may be widely varied depending on the needs and characteristics of a particular system. By way of example, for a virtual machine intended to provide high execution performance, the predetermined maximum level may be within the range of approximately 10 percent to approximately 30 percent usage of the overall system resources. When the overhead associated with the compilation of byte codes 144 exceeds the predetermined maximum level, then methods which would be compiled if the compilation overhead were lower may be placed in an execution list, or queue. This execution list is essentially a waiting list of methods which may be processed when the compilation overhead is lower, *i.e.*, methods in the execution list may be compiled when the compilation overhead is lower. Specifically, the execution list may be processed at times when both the compilation overhead, as well as the overall computation overhead associated with computer system 146, are at a low level.

When byte codes are to be compiled during otherwise idle periods in the overall execution of a program, *e.g.*, during idle periods which occur throughout the processing of byte codes, the program may be monitored to determine when pauses

occur. Referring next to Figure 2, a process of executing a program, or an application, which includes the capability of compiling code during pauses will be described in accordance with an embodiment of the present invention. The execution of a program begins at step 210, and in step 212, a determination is made regarding whether the execution of the program has reached an idle period.

An idle period is generally a pause, *e.g.*, a think pause, in the processing of the program during which the program is essentially inactive. The program may be awaiting an action such as a timer signal from the associated operating system, or the program may be awaiting a user input, prior to actively continuing execution. In general, a think pause occurs while the program is awaiting an action from a user. It should be appreciated that the length of an idle period may vary widely. By way of example, the length of an idle period may vary from very few milliseconds to approximately half a second. When an idle period occurs because a program is awaiting input from a user, the idle period may generally be much longer, as for example on the order of a number of seconds, minutes, days or more.

In general, a threshold value is used to recognize a "useful" pause. For example, if the actual pause has lasted for a predetermined number of milliseconds, then the system may recognize the pause as a useful pause which is likely to continue for a whole longer. When the pause is likely to continue, then compilations may be allowed to continue in order to exploit the pause. As such, the determination in step 212 regarding whether the execution of the program has reached an idle period

involves determining whether a period of low activity has reached a threshold value, *e.g.*, approximately 100 milliseconds or approximately half a second.

In general, the occurrence of an idle period, or period of relatively low activity, may be identified by monitoring the usage of the central processing unit (CPU) associated with the execution of the program. Alternatively, the existence of an idle period may be determined by monitoring the status of threads associated with the overall computer system.

When CPU usage is monitored to identify periods of relative inactivity, CPU usage is essentially monitored to determine when the CPU usage falls below a certain usage threshold. The usage threshold, *i.e.*, a "low activity threshold," may be widely varied. By way of example, the usage threshold may be set such that when substantially only activities with low overhead, such as setting a cursor to periodically blink, are running, the program is considered to be idle. In one embodiment, when CPU usage during the processing of the program falls below a usage threshold which is approximately 20 percent of the overall system resources, then the overall processing of the program is considered to be in a period of low activity. At such times, at least approximately 80 percent of the system resources may be available for use.

Monitoring the status of threads may involve studying a thread scheduler, as will be appreciated by those skilled in the art. When the thread scheduler indicates that all threads are blocked, *i.e.*, substantially none of the threads are in a "run-able"

state, then the implication is that the program is not consuming CPU time. CPU time will generally not be consumed until a signal arrives, at which point a thread may become unblocked. When CPU time is not being consumed, then the program is generally in an idle period.

If it is determined in step 212 that the execution of the program is not in an idle period, then process flow returns to step 210 where the program continues to execute. If, instead, it is determined in step 212 that there is an idle period, a determination is made in step 214 regarding whether there are any pending tasks, as for example compilations. That is, a determination is made regarding whether there are any tasks which are slated to occur during idle periods. The tasks may generally include a wide variety of tasks, as for example compilations and garbage collection. Such tasks may be prioritized in terms of their relative importance within a system. In the described embodiment, the tasks will be described in terms of pending compilations.

Pending compilations may generally be obtained from two sources, a database list and an execution list, which are lists of "candidate" methods for compilation. One arrangement for creating a database list may involve a system which monitors compiled methods, such as an arrangement for creating a database list when a database is inlined. In such a system, the database list is essentially a "working set," or a "candidate set," of methods which were compiled during a previous execution of the program, and are considered as methods that are likely to be compiled in a subsequent execution of the program. Other mechanisms may be used to create or

add to a database list as well. It should be appreciated that in some embodiments, the database list may include other tasks such as the performance of garbage collection.

An execution list, as mentioned above, is a queue of methods, or a candidate set of methods, which is created during the current execution of the program. The execution list may contain, or identify, methods which were suppressed from compilation due to the fact that the compilation overhead was previously considered to be too high to allow the methods to be compiled. The execution list may also identify methods which were periodically added to the list by a separate sweeper process which studies the invocation counters, *i.e.*, counters which track the number of times a method has been interpreted, associated with interpreted methods. In general, the invocation counter for a method is incremented each time the method is invoked. The sweeper may periodically study invocation counters to determine if invocation counters of methods which have not been recently invoked are at a level which is currently considered to deem the method as appropriate for compilation. Since the threshold, *e.g.*, the number of invocations of a method which must generally be reached in order for the method to be considered for compilation, may vary during program execution, methods with invocation counters that are over a current threshold may be added to the execution list by the sweeper. Other mechanisms may be used to create or add to the execution list as well. In some embodiments, the execution list may be associated with a general work list which includes pending compilations in addition to other pending tasks, *e.g.*, garbage collection.

When it is determined through studying the execution list and the database list in step 214 that there are no pending tasks, then process flow returns to step 210 in which the program continues to execute. When, on the other hand, it is determined that there are pending compilations, then in step 216, the tasks, *e.g.*, compilations, are performed. The steps associated with performing the pending compilations will be described below with reference to Figure 3. After pending compilations are performed, process flow returns to step 210 in which the program continues to execute.

As previously mentioned, in the described embodiment, a method is compiled when it is anticipated that compiling the method and invoking the compiled method is likely to be more efficient than interpreting the method. Such compilations may be delayed until there are idle periods during the execution of the overall program, in order to avoid excessive compilation overhead. Figure 3 is a process flow diagram which illustrates the steps associated with performing delayed compilations on methods, *i.e.*, step 216 of Figure 2, in accordance with an embodiment of the present invention. The process of performing compilations begins at step 302 where the candidate method with the highest priority is identified. In one embodiment, the highest priority method may be the method with the highest number of invocation counts in, or identified by, the execution list. Alternatively, the highest priority method may be the method with the shortest anticipated compilation time on the execution list. The identification of the highest priority method will be discussed in more detail below with reference to Figure 4.

After the highest priority method is identified in step 302, a determination is made in step 304 regarding whether the highest priority method has already been compiled. The highest priority method may already be compiled for a variety of different reasons. By way of example, after the method is placed in the execution list, during the course of overall program execution, the method may be invoked and compiled while compilation overhead is low. As such, the method is already compiled, and may not have been removed from the execution list, as repeatedly updating the execution list is expensive. The highest priority method may also be previously compiled if the method is obtained from the database list. That is, when the same method appears in both the execution list and the database list, if the method was already compiled using the execution list, for example, and the database list is not updated due to high costs, then the method may already be compiled when it is identified as the highest priority method in the database list.

When the determination in step 304 is that the highest priority method is already compiled, then the method may be removed from the appropriate list, *i.e.*, either the execution list or the database list, and process flow returns to step 302 where a new highest priority method is identified. If, instead, it is determined that the highest priority method has not been previously compiled, then in step 306, the compilation of the highest priority method begins. That is, the compilation of the highest priority method is initialized.

During the compilation of the highest priority method, interrupts may be received. In general, interrupts may include, but are not limited to, timer signals and

user inputs, *e.g.*, keyboard inputs. It is determined in step 308 whether an interrupt has been received. As will be appreciated by those skilled in the art, when an interrupt is received, the compilation of the highest priority method may automatically cease in order for the interrupt to be processed as necessary.

If an interrupt has been received, then the compilation of the highest priority method is allowed to continue for a predetermined period of time in step 314. The predetermined period of time is, in one embodiment, essentially a "time to live" for a compilation after a signal has been received that will set at least one thread in a run-able state. By allowing the compilation to continue briefly after an interrupt is received, the compilation is given a chance to be completed, in the event that the compilation is close to completion.

Although the predetermined period of time may be widely varied, the predetermined period of time is generally a number of milliseconds, *e.g.*, "X milliseconds." Shorter periods of time in the range of approximately 10 milliseconds to approximately 200 milliseconds are preferred, since it has been observed that delays of time of greater than approximately 200 milliseconds in response to user input are perceptible and, hence, often annoying to a user. The period of time may be varied based on the source of the signal or the priority of the threads in a run-able state, with shorter periods being used for higher-priority signals or threads.

After the compilation is allowed to continue for the predetermined period of time in step 314, a determination is made in step 316 regarding whether the

compilation has been successfully completed during the predetermined period of time. If the compilation has been completed, then process flow moves to step 210 of Figure 2, where the overall execution of the program continues. It should be appreciated that once the compilation is completed, the newly compiled method will also be removed from either the execution list or the database list, depending upon which list the method was obtained from.

When the compilation is not completed during the predetermined period of time, then process flow moves from step 316 to step 318 in which the compilation is aborted. The compilation is not allowed to be completed due to the fact that allowing the compilation to be completed may result in a significant delay in the execution of the overall program or, more specifically, the processing of the interrupt. When the compilation is aborted, "clean up" occurs, *e.g.*, system resources used in the attempted compilation are relinquished, as will be understood by those skilled in the art. Once the compilation is aborted, process flow moves to step 210 of Figure 2, in which the overall execution of the program continues.

Returning to step 308, in the event that no interrupt has been received during the compilation of the highest priority method, the compilation continues in step 310. A determination is made in step 312 regarding whether the compilation has been completed. If it is determined that the compilation has been completed, the highest priority method may be removed from the appropriate list, and process flow returns to step 302 in which a new highest priority method is identified. Alternatively, if it is determined in step 312 that the compilation has not been completed, then the

compilation is allowed to continue until either it is completed, or it is aborted due to the receipt of an interrupt in step 308.

In general, the "order" in which methods in an execution list and a database list are compiled may be determined based upon a number of different factors. The methods in each list may be assigned priority values relative to other methods in the same list. The priority values may generally be assigned at any time during the overall execution of the program, and may further be updated throughout the course of program execution. By way of example, the priority values in a list may be computed and updated, *i.e.*, re-evaluated, each time a method is added to the list. Priority values may also be re-evaluated at the beginning of each idle period to ensure that the priorities of methods in the execution list and the database list are current at the time methods are chosen for compilation.

The priority value of a method may be calculated based on any combination of factors including, but not limited to, the invocation count of the method, the position of the method within the list, the estimated compilation time of the method, and the length of the current idle period during the overall execution of the program. Alternatively, in one embodiment, the priority value of a method may essentially be assigned randomly.

The method which has the highest invocation count in a list may have the highest compilation priority, as it is essentially the most frequently invoked interpreted method. The position of a method within a list may also affect the priority

value assigned to the method, since the method most recently added to the list is often likely to be needed in the immediate future. The method which has the shortest estimated compilation time in a list may have the highest compilation priority, since the compilation of a method with a shorter estimated compilation time is generally more likely to be completed during an idle period than the compilation of a method with a longer estimated compilation time. In general, the compilation time of a method may be estimated by the length of the method, *e.g.*, a relatively short method will typically have a relatively short compilation time.

The length of the current idle period may also affect the priority value of a method. For example, if the current idle period has already been relatively long, then the likelihood that the idle period will last longer is generally increased. Long idle periods may often be a result of a user leaving the computer on which the overall execution of the program is occurring for a prolonged period of time. As such, it has been observed that when it is determined that a current idle period has already been relatively long, the likelihood that the idle period will last even longer is generally reasonably high. Therefore, when an idle period is expected to be relatively long, methods with longer expected compilation times may have higher compilation priorities, since methods with shorter expected compilation times may typically be readily compiled during subsequent shorter idle periods.

As previously mentioned, the priority value of a method may be based on a number of different factors. In other words, a priority function may be used to determine a priority value. The form of such a priority function may generally be

widely varied. By way of example, a priority function may divide the invocation count for a method by the estimated compilation time for the method in order to obtain a priority value for the method. Alternatively, a priority function may involve using the invocation count for methods with shorter estimated compilation times and using the invocation count divided by a constant for methods with longer estimated compilation times.

Referring next to Figure 4, the steps associated with identifying the method with the highest priority value, *i.e.*, step 302 of Figure 3, will be described in accordance with an embodiment of the present invention. The identification of the method with the highest priority begins at step 402 with a determination of whether any methods are present in the execution list associated with the program.

In general, methods in an execution list are considered as having compilation priorities that are higher than those of methods in a database list. Methods in the execution list are methods that have been identified during the course of program execution as being likely to overcome the associated compilation overhead by executing more efficiently, because the methods are repeatedly invoked. Alternatively, methods in the database list are methods which may potentially be repeatedly invoked and, hence, compiling the methods is likely to be beneficial. In other words, the execution list reflects the current, or actual, usage of methods, while the database list essentially reflects long-term usage of methods. Therefore, methods from the execution list are typically chosen for compilation before methods from the

database list, since compiling the methods in the execution list is more likely to lead to a more efficient overall execution of the program which uses the methods.

If the determination in step 402 is that there are methods in the execution list, then in step 404, the method with the highest priority in the execution list is selected for compilation. Once the method is selected from the execution list, then the process of identifying the highest priority method is completed. Alternatively, if the determination in step 402 is that there are no methods in the execution list, then the highest priority method in the database list is selected for compilation. After the method is selected from the database list, then the process of identifying the highest priority method is completed.

The present invention may generally be implemented on any suitable computer system. Specifically, the compilation of queued methods during idle periods in the overall execution of a program may be accomplished using any suitable virtual machine, such as the virtual machine described below with respect to Figure 6. Figure 5 illustrates a typical, general purpose computer system suitable for implementing the present invention. The computer system 530 includes any number of processors 532 (also referred to as central processing units, or CPUs) that are coupled to memory devices including primary storage devices 534 (typically a read only memory, or ROM) and primary storage devices 536 (typically a random access memory, or RAM).

Computer system 530 or, more specifically, CPU 532, may be arranged to support a virtual machine, as will be appreciated by those skilled in the art. One example of a virtual machine that is supported on computer system 530 will be described below with reference to Figure 6. As is well known in the art, ROM acts to transfer data and instructions uni-directionally to the CPU 532, while RAM is used typically to transfer data and instructions in a bi-directional manner. CPU 532 may generally include any number of processors. Both primary storage devices 534, 536 may include any suitable computer-readable media. A secondary storage medium 538, which is typically a mass memory device, is also coupled bi-directionally to CPU 532 and provides additional data storage capacity. The mass memory device 538 is a computer-readable medium that may be used to store programs including computer code, data, and the like. Typically, mass memory device 538 is a storage medium such as a hard disk or a tape which is generally slower than primary storage devices 534, 536. Mass memory storage device 938 may take the form of a magnetic or paper tape reader or some other well-known device. It will be appreciated that the information retained within the mass memory device 538, may, in appropriate cases, be incorporated in standard fashion as part of RAM 534 as virtual memory. A specific primary storage device 536such as a CD-ROM may also pass data uni-directionally to the CPU 532.

CPU 532 is also coupled to one or more input/output devices 540 that may include, but are not limited to, devices such as video monitors, track balls, mice, keyboards, microphones, touch-sensitive displays, transducer card readers, magnetic or paper tape readers, tablets, styluses, voice or handwriting recognizers, or other

well-known input devices such as, of course, other computers. Finally, CPU 532 optionally may be coupled to a computer or telecommunications network, *e.g.*, a local area network, an internet network or an intranet network, using a network connection as shown generally at 512. With such a network connection, it is contemplated that the CPU 532 might receive information from the network, or might output information to the network in the course of performing the above-described method steps. Such information, which is often represented as a sequence of instructions to be executed using CPU 532, may be received from and outputted to the network, for example, in the form of a computer data signal embodied in a carrier wave. The above-described devices and materials will be familiar to those of skill in the computer hardware and software arts.

As previously mentioned, a virtual machine may execute on computer system 530. Figure 6 is a diagrammatic representation of a virtual machine which is supported by computer system 530 of Figure 5, and is suitable for implementing the present invention. When a computer program, *e.g.*, a computer program written in the Java™ programming language developed by Sun Microsystems of Mountain View, California, is executed, source code 610 is provided to a compiler 620 within a compile-time environment 605. Compiler 620 translates source code 610 into byte codes 630. In general, source code 610 is translated into byte codes 630 at the time source code 610 is created by a software developer.

Byte codes 630 may generally be reproduced, downloaded, or otherwise distributed through a network, *e.g.*, network 512 of Figure 5, or stored on a storage

device such as primary storage 534 of Figure 5. In the described embodiment, byte codes 630 are platform independent. That is, byte codes 630 may be executed on substantially any computer system that is running a suitable virtual machine 640. By way of example, in a Java™ environment, byte codes 630 may be executed on a computer system that is running a Java™ virtual machine.

Byte codes 630 are provided to a runtime environment 635 which includes virtual machine 640. Runtime environment 635 may generally be executed using a processor such as CPU 532 of Figure 5. Virtual machine 640 includes a compiler 642, an interpreter 644, and a runtime system 646. Byte codes 630 may generally be provided either to compiler 642 or interpreter 644.

When byte codes 630 are provided to compiler 642, methods contained in byte codes 630 are compiled into machine instructions, as described above. On the other hand, when byte codes 630 are provided to interpreter 644, byte codes 630 are read into interpreter 644 one byte code at a time. Interpreter 644 then performs the operation defined by each byte code as each byte code is read into interpreter 644. In general, interpreter 644 processes byte codes 630 and performs operations associated with byte codes 630 substantially continuously.

When a method is called from an operating system 660, if it is determined that the method is to be invoked as an interpreted method, runtime system 646 may obtain the method from interpreter 644. If, on the other hand, it is determined that the method is to be invoked as a compiled method, runtime system 646 activates

compiler 642. Compiler 642 then generates machine instructions from byte codes 630, and executes the machine-language instructions. In general, the machine-language instructions are discarded when virtual machine 640 terminates.

Although only a few embodiments of the present invention have been described, it should be understood that the present invention may be embodied in many other specific forms without departing from the spirit or the scope of the invention. By way of example, steps involved with performing a compilation during an idle period may be reordered, removed or added. In general, steps involved with the methods of the present invention may be reordered, removed, or added without departing from the spirit or the scope of the present invention.

A highest priority method for compilation has been described as being selected out of substantially all methods in an execution list and a database list. Searching all methods in an execution list or a database list, or both, to locate the method with the highest compilation priority may prove to be expensive in the event that the execution list is extensive. Therefore; in one embodiment, a subset of the execution list may be searched, in lieu of substantially the entire execution list, to identify a highest priority method for compilation may. By way of example, the first "N" candidate methods from the execution list may be searched to identify the highest priority method in the first N candidate methods.

As previously mentioned, a compilation priorities of methods may be identified, at least in part, using the invocation counters. The invocation counters are

generally counters which are incremented each time a method is accessed. It should be appreciated that in some embodiments, invocation counters may be decayed over time. For example, if a method was repeatedly invoked and slated for compilation at the beginning of program execution, but is never again invoked, the invocation counter may be decayed to reduce the compilation priority of the method. Once the invocation counter for a method has been decayed, *e.g.*, exponentially decayed, compiling the method may no longer be advisable. As such, such methods may be periodically removed from the execution list.

Similarly, during the course of performing compilations, a determination is made regarding whether a method which is slated for compilation has previously been compiled. Alternatively, methods which have already been compiled may be removed as appropriate from either or both the execution list and the database list. In general, the overhead associated with checking the execution list and the database list for compiled methods may be relatively significant. However, compiled methods may be periodically removed, as for example at the beginning of an idle period, without departing from the spirit or the scope of the present invention.

Although an execution list has generally been described as being a list, or queue, of interpreted methods which are to be compiled, the execution list may also include compiled methods which are delayed from being re-compiled until there is an idle period. For example, in a system with two levels of compilers, once a method is compiled, if that method is repeatedly executed, it may prove to be desirable for the method to be re-compiled more efficiently. In such a system, the methods slated for

re-compilations, or secondary compilations, may be included in the same execution list as interpreted methods which are slated for compilation. However, the methods slated for re-compilations may, on the other hand, be included in a separate execution list.

In one embodiment, the execution list may generally be a work list which includes delayed tasks, in addition to, or in lieu of, pending compilations. Such tasks may be prioritized. It should be appreciated that when a task from the work list is running, if an interrupt is received, the task may be allowed to be completed. By way of example, if the task involves garbage collection, suspending a garbage collection process and "backing out" of the garbage collection process may be as costly, if not more costly, than completing the process. As such, the garbage collection may be allowed to be completed. Tasks may also generally be suspended when an interrupt is received. Therefore, the present examples are to be considered as illustrative and not restrictive, and the invention is not to be limited to the details given herein, but may be modified within the scope of the appended claims along with their full scope of equivalents.

4. Brief Description of Drawings

Figure 1 is a block diagram representation of a computer system which dynamically compiles code in accordance with an embodiment of the present invention.

Figure 2 is a process flow diagram which illustrates the steps associated with executing a program which includes the capability of compiling code during pauses in accordance with an embodiment of the present invention.

Figure 3 is a process flow diagram which illustrates the steps associated with performing compilations on methods, *i.e.*, step 216 of Figure 2, in accordance with an embodiment of the present invention.

Figure 4 is a process flow diagram which illustrates the steps associated with identifying the highest priority method, *i.e.*, step 302 of Figure 3, in accordance with an embodiment of the present invention.

Figure 5 illustrates a typical, general purpose computer system suitable for implementing the present invention.

Figure 6 is a diagrammatic representation of a virtual machine which is suitable for implementing the present invention.

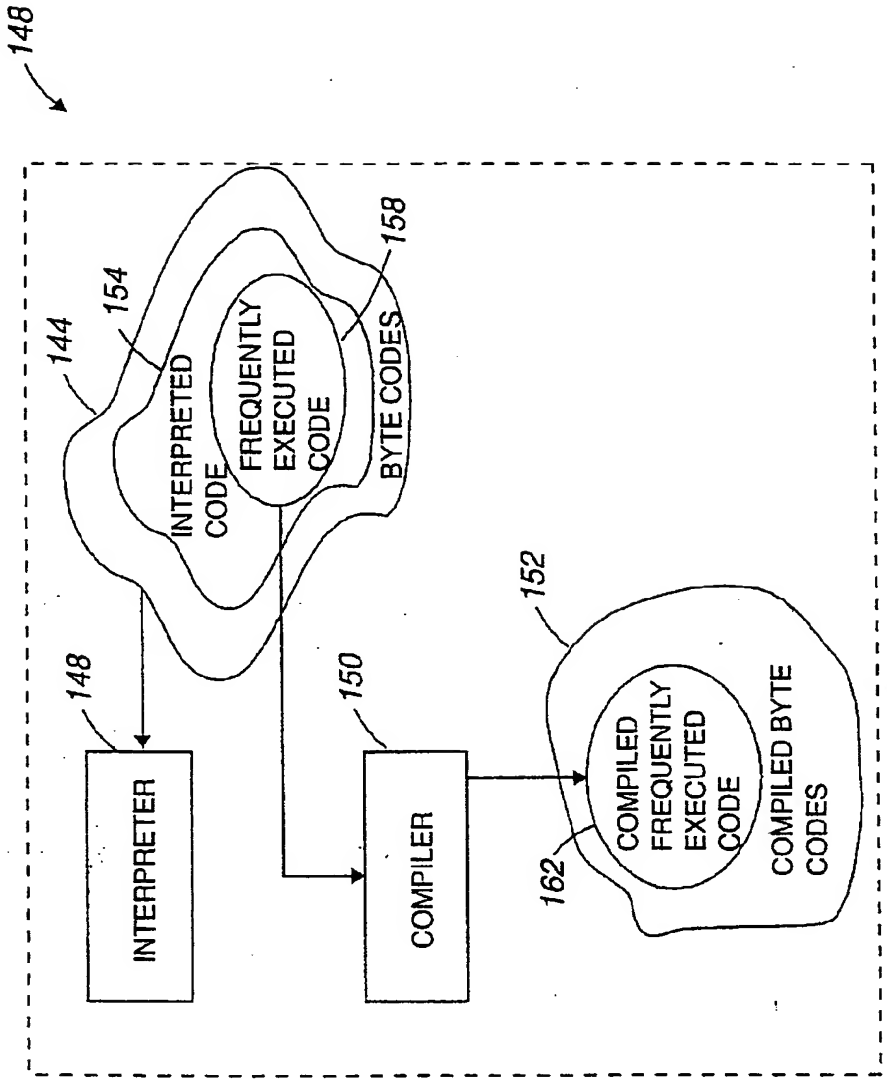

148
144
154
158
INTERPRETED CODE
FREQUENTLY EXECUTED CODE
BYTE CODES
148
INTERPRETER
150
COMPILER
152
COMPILED FREQUENTLY EXECUTED CODE
COMPILED BYTE CODES
162

Figure 1

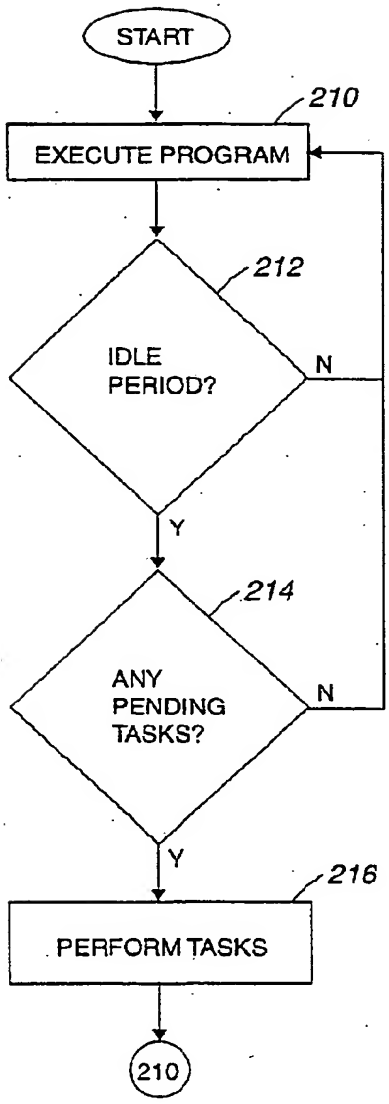

START
210
EXECUTE PROGRAM
212
IDLE PERIOD?
N
Y
214
ANY PENDING TASKS?
N
Y
216
PERFORM TASKS
210

Figure 2

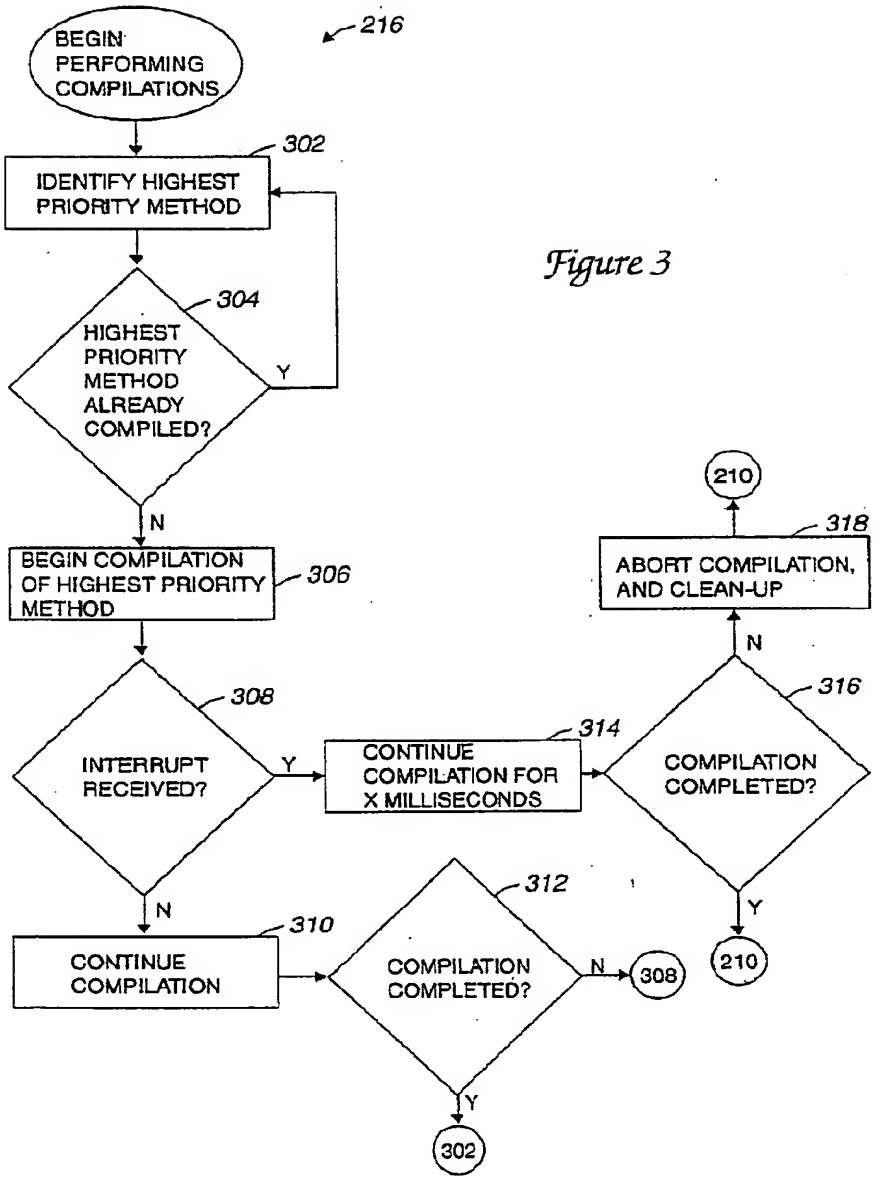

216
BEGIN PERFORMING COMPILATIONS
302
IDENTIFY HIGHEST PRIORITY METHOD
304
HIGHEST PRIORITY METHOD ALREADY COMPILED?
Y
N
BEGIN COMPILATION OF HIGHEST PRIORITY METHOD
306
308
INTERRUPT RECEIVED?
Y
314
CONTINUE COMPILATION FOR X MILLISECONDS
316
COMPILATION COMPLETED?
N
318
ABORT COMPILATION, AND CLEAN-UP
210
Y
210
N
310
CONTINUE COMPILATION
312
COMPILATION COMPLETED?
N
308
Y
302

Figure 3

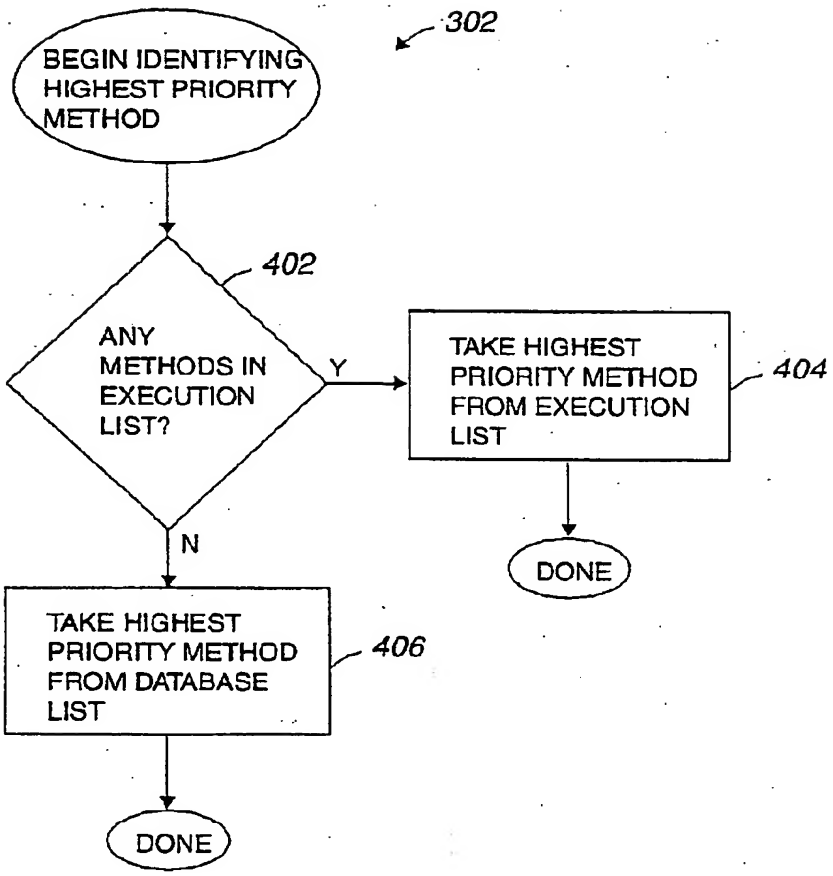

302
BEGIN IDENTIFYING HIGHEST PRIORITY METHOD
402
ANY METHODS IN EXECUTION LIST?
Y
TAKE HIGHEST PRIORITY METHOD FROM EXECUTION LIST
404
DONE
N
TAKE HIGHEST PRIORITY METHOD FROM DATABASE LIST
406
DONE

Figure 4

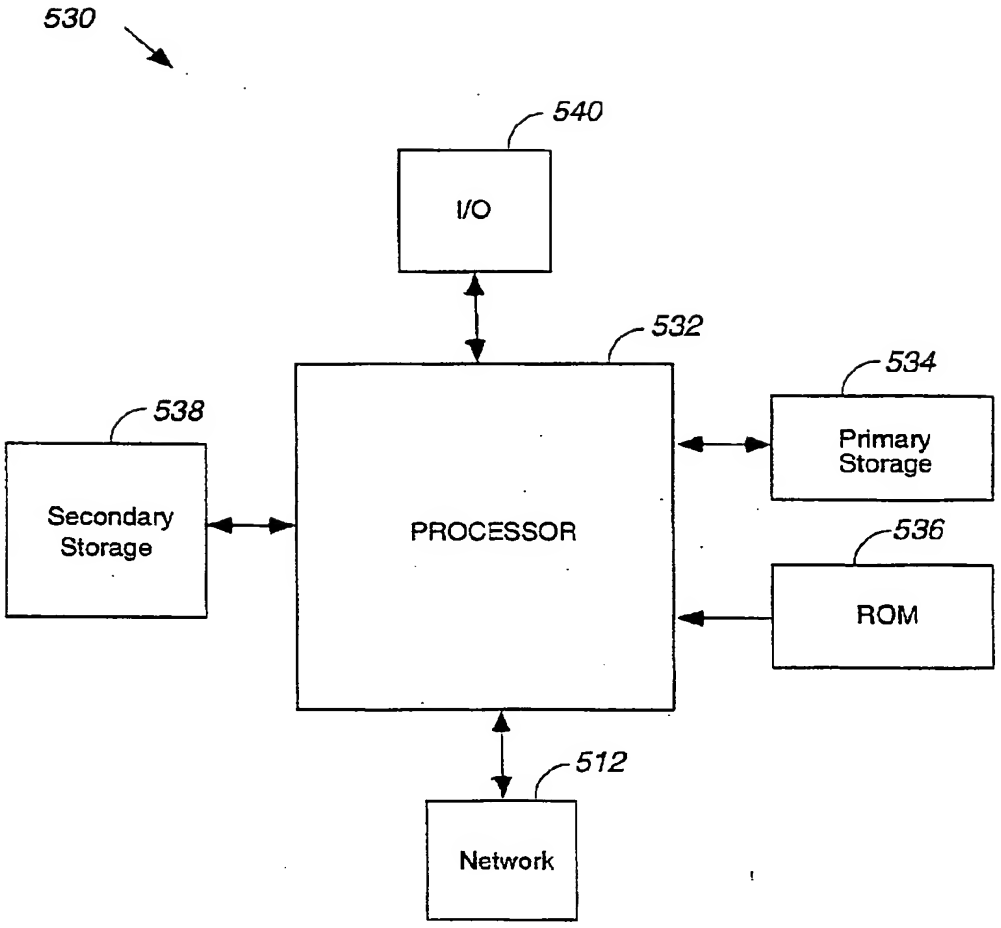

530
540
I/O
532
534
538
Secondary Storage
PROCESSOR
Primary Storage
536
ROM
512
Network

Figure 5

整理番号：P98BW-006 (6/6)

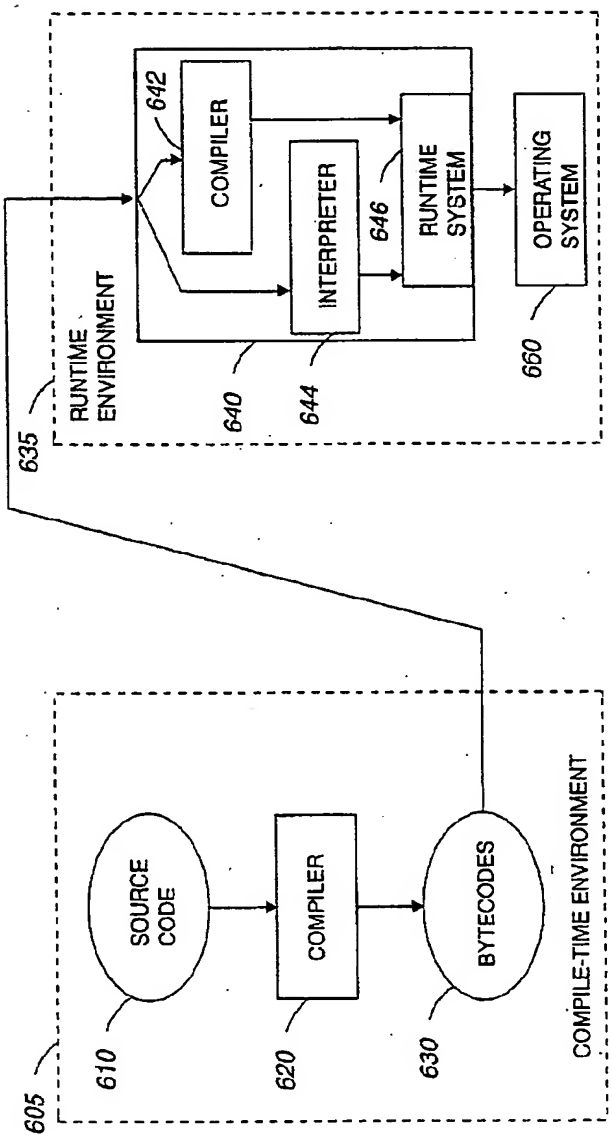


Figure 6

1. Abstract

Methods and apparatus for dynamically compiling byte codes associated with methods during idle periods in the execution of a computer program are disclosed. The described methods are particularly suitable for use in computer systems that are arranged to execute both interpreted and compiled byte codes. In some embodiments, methods to be dynamically compiled are referenced in one or more lists. The lists may be prioritized to facilitate the compilation of the highest priority methods first. In one embodiment, a pair of compilation lists are provided with a first one of the compilation lists being created prior to processing the computer program while the other is created during the processing of the computer program.

2. Representative Drawing

Fig. 2